

conference

proceedings

LISA XVI: 16th Systems Administration Conference

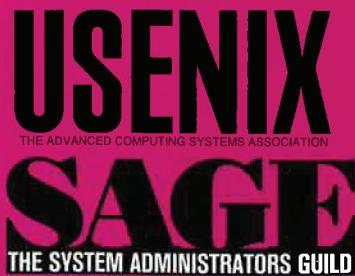
Philadelphia, Pennsylvania Nov 3-8, 2002

LISA XVI

**Sixteenth Systems
Administration
Conference**

*Philadelphia, Pennsylvania, USA
November 3–8, 2002*

Sponsored by **The USENIX Association** and
SAGE, the System Administrators Guild



For additional copies of these proceedings contact

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Telephone: +1 510-528-8649
<http://www.usenix.org>
<office@usenix.org>

The price is \$35 for members and \$45 for nonmembers.
Outside the U.S.A. and Canada, please add \$18 per copy for postage
(via air printed matter).

Past LISA Conferences

Large Installation Systems Admin. I Workshop	1987	Philadelphia, PA
Large Installation Systems Admin. II Workshop	1988	Monterey, CA
Large Installation Systems Admin. III Workshop	1989	Austin, TX
Large Installation Systems Admin. IV Conference	1990	Colorado Spgs, CO
Large Installation Systems Admin. V Conference	1991	San Diego, CA
Systems Administration VI Conference	1992	Long Beach, CA
Systems Administration VII Conference	1993	Monterey, CA
Systems Administration VIII Conference	1994	San Diego, CA
Systems Administration IX Conference	1995	Monterey, CA
Systems Administration X Conference	1996	Chicago, IL
Systems Administration XI Conference	1997	San Diego, CA
Systems Administration XII Conference	1998	Boston, MA
Systems Administration XIII Conference	1999	Seattle, WA
Systems Administration XIV Conference	2000	New Orleans, LA
Systems Administration XV Conference	2001	San Diego, CA

Copyright © 2002 by The USENIX Association. All rights reserved.

This volume is published as a collective work.

Rights to individual papers remain with the author or the author's employer.

Permission is granted for the noncommercial reproduction
of the complete work for educational or research purposes.

USENIX acknowledges all trademarks appearing herein.

ISBN 1-931971-03-X

 Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

USENIX Association

**Proceedings of the Sixteenth
Systems Administration Conference
(LISA XVI)**

**November 3-8, 2002
Philadelphia, PA, USA**

CONTENTS

Author Index	v
Acknowledgments	vii
Preface	ix

Opening Remarks

Wednesday (8:45-9:00 am)

Chair: Alva Couch

Keynote

Wednesday (9:00-10:30 am)

Speaker: Jim Reese

Working Smarter

Wednesday (11:00 am-12:30 pm)

Chair: Aileen Frisch

Work-Augmented Laziness with the Los Task Request System	1
<i>Thomas Stepleton, Swarthmore College Computer Society</i>	
Spam Blocking with a Dynamically Updated Firewall Ruleset	13
<i>Deeann M. M. Mikula, Chris Tracy, and Mike Holling, Telerama Public Access Internet</i>	
Holistic Quota Management: The Natural Path to a Better, More Efficient Quota System	21
<i>Michael Gilfix, Tufts University</i>	

Service, Risk, and Scale

Wednesday (2:00-3:30 pm)

Chair: Alex Keller

Application Aware Management of Internet Data Center Software	33
<i>Alain Mayer, CenterRun, Inc.</i>	
Geographically Distributed System for Catastrophic Recovery	47
<i>Kevin Adams, NSW CDD</i>	
Embracing and Extending Windows 2000	65
<i>Jon Finke, Rensselaer Polytechnic Institute</i>	

Practical Theory

Wednesday (4:00-5:30 pm)

Chair: Paul Anderson

Stem: The System Administration Enabler	75
<i>Uri Guttman, Stem Systems, Inc.</i>	
Pan: A High-Level Configuration Language	83
<i>Lionel Cons and Piotr Poznański, CERN, European Organization for Nuclear Research</i>	
Why Order Matters: Turing Equivalence in Automated Systems Administration	99
<i>Steve Traugott, TerraLuna, LLC; Lance Brown, National Institute of Environmental Health Sciences</i>	

Logging and Monitoring

Thursday (9:00-10:30 am)

Chair: Marcus Ranum

- A New Architecture for Managing Enterprise Log Data 121
Adam Sah, Addamark Technologies, Inc.

- MieLog: A Highly Interactive Visual Log Browser Using Information Visualization and Statistical Analysis 133
Tetsuji Takada & Hideki Koike, University of Electro-Communications

- Process Monitor: Detecting Events That Didn't Happen 145
Jon Finke, Rensselaer Polytechnic Institute

Short Subjects

Thursday (11:00 am-12:30 pm)

Chair: Alva Couch

- An Analysis of RPM Validation Drift 155
John Hart and Jeffrey D'Amelia, Tufts University

- RTG: A Scalable SNMP Statistics Architecture for Service Providers 167
Robert Beverly, MIT Laboratory for Computer Science

- Environmental Acquisition in Network Management 175
Mark Logan, Matthias Felleisen, and David Blank-Edelman, Northeastern University

- A Simple Way to Estimate the Cost of Downtime 185
David A. Patterson, University of California at Berkeley

Service and Network Upgrades

Thursday (2:00-3:30 pm)

Chair: Steve Traugott

- Defining and Monitoring Service Level Agreements for Dynamic e-Business 189
Alexander Keller and Heiko Ludwig, IBM T. J. Watson Research Center

- HotSwap – Transparent Server Failover for Linux 205
Noel Burton-Krahn, HotSwap Network Solutions

- Over-Zealous Security Administrators Are Breaking the Internet 213
Richard van den Berg, Trust Factory b.v.; Phil Dibowitz, University of Southern California

Security

Friday (9:00-10:30 am)

Chair: Marcus Ranum

- An Approach for Secure Software Installation 219
V. N. Venkatakrishnan, R. Sekar, T. Kamat, S. Tsipa and Z. Liang, Computer Science Dept., SUNY at Stony Brook

- Network-based Intrusion Detection – Modeling for a Larger Picture 227
Atsushi Totsuka, Tohoku University; Hidenari Ohwada, NTT, Tokyo; Nobuhisa Fujita, Tohoku University; Debasish Chakraborty, Tohoku University; Glenn Mansfield Keeni, Cyber Solutions, Inc.; Norio Shiratori, Tohoku University

- Timing the Application of Security Patches for Optimal Uptime 233
Steve Beattie, Seth Arnold, Crispin Cowan, Perry Wagle, and Chris Wright, WireX Communications, Inc.; Adam Shostack, Zero Knowledge Systems, Inc.

AUTHOR INDEX

Kevin Adams	47	Z. Liang	219
Seth Arnold	233	Mark Logan	175
Steve Beattie	233	Heiko Ludwig	189
Robert Beverly	167	Alain Mayer	33
David Blank-Edelman	175	Deeann M. M. Mikula	13
Lance Brown	99	Hidenari Ohwada	227
Noel Burton-Krahn	205	David A. Patterson	185
Debasish Chakraborty	227	Piotr Poznański	83
Lionel Cons	83	Adam Sah	121
Crispin Cowan	233	R. Sekar	219
Jeffrey D'Amelia	155	Norio Shiratori	227
Phil Dibowitz	213	Adam Shostack	233
Matthias Felleisen	175	Thomas Stepleton	1
Jon Finke	65, 145	Tetsuji Takada	133
Nobuhisa Fujita	227	Atsushi Totsuka	227
Michael Gilfix	21	Chris Tracy	13
Uri Guttman	75	Steve Traugott	99
John Hart	155	S. Tsipa	219
Mike Holling	13	Richard van den Berg	213
T. Kamat	219	V. N. Venkatakrishnan	219
Glenn Mansfield Keeni	227	Perry Wagle	233
Alexander Keller	189	Chris Wright	233
Hideki Koike	133		

ACKNOWLEDGMENTS

PROGRAM CHAIR

Alva Couch, *Tufts University*

PROGRAM COMMITTEE

Paul Anderson, *University of Edinburgh*
Robert Aphorpe
Aleen Frisch, *Exponential Consulting*
Alexander Keller, *IBM Research*
Tom Limoncelli, *Lumeta Corp.*
Will Partain, *Verilab*
David Parter, *University of Wisconsin*
Marcus Ranum, *Ranum.com*
John Sellens, *Certainty Solutions*
Josh Simon, *ATG*
Steve Traugott, *TerraLuna, Inc.*

COPY EDITOR

Jeff Allen, *Tellme Networks*

SCRIBE

Adam Moskowitz

NETWORK TRACK COORDINATOR

David Williamson, *Certainty Solutions*

SECURITY TRACK COORDINATOR

Lynda True, *TRW, Inc.*

GURU-IS-IN COORDINATOR

Lee Damon, *University of Washington*

REFEREED PAPER EXTERNAL REVIEWERS

Jeff R. Allen, *Tellme Networks, Inc.*
Lee Amatangelo
Martin Andrews, *LION Bioscience, Inc.*
Kenytt Avery, *Willing Minds LLC*
Rocky Bernstein, *About.com*
Mark Burgess, *Oslo Univ. College*
Crispin Cowan, *WireX*
Sven Dietrich, *CERT Coordination Ctr.*
Chris Faehl, *RightNow Technologies*
Adrian Filipi-Martin, *Tovaris, Inc.*
Luc Girardin, *Macrofocus GmbH*
David Harnick-Shapiro, *Univ. of California at Irvine*
Doug Hughes, *Global Crossing*
Philip Kizer, *Texas A&M Univ.*
L. F. Marshall, *Univ. of Newcastle*
Patrick McCormick, *Tellme Networks, Inc.*

INVITED TALKS COORDINATORS

Strata Rose Chalup, *VirtualNet Consulting*
Esther Filderman, *Pitt. Supercomputing Ctr.*

WORK-IN-PROGRESS COORDINATOR

Peg Schafer

CONFERENCE ADVISOR

Rob Kolstad, *SAGE*

METALISA WORKSHOP

Cat Okita, *Earthworks*
Tom Limoncelli, *Lumeta Corp.*

CONFIGURATION WORKSHOP

Paul Anderson, *University of Edinburgh*
Alva Couch, *Tufts University*

CURRICULUM/TAXONOMY WKSHOP

Rob Kolstad, *SAGE*
John Sechrest, *PEAK, Inc.*
Curt Freeland, *Notre Dame University*

AFS WORKSHOP

Esther Filderman, *Pitts. Supercomputing Ctr.*
Derrick Brashear, *Carnegie Mellon University*

ADVANCED TOPICS WORKSHOP

Adam Moskowitz
Rob Kolstad, *SAGE*

Ellen Mitchell, *Texas A&M Univ.*

Adam Moskowitz

Tejas Naik, *Lucent Technologies*

Ben Nelson, *RightNow Technologies*

David A. Patterson, *Univ. of California at Berkeley*

Kathryn L. Penn, *Univ. of Maryland*

S. Raj Rajagopalan, *Telcordia Technologies*

Frode Eika Sandnes, *Oslo Univ. College*

Jürgen Schönwälder, *Univ. of Osnabrück, Germany*

Vlakkies Schreuder, *Principia Mathematica Inc*

Demosthenes Skipitaris, *Fast Search & Transfer*

Jon Stearley, *Compaq/HP*

Sigmund Straumsnes, *Oslo Univ. College*

Todd K. Watson, *Southwestern Univ.*

Tara Whalen, *Communications Research Centre Canada*

PREFACE

On behalf of the Program Committee, conference organizers, USENIX, and SAGE, welcome to LISA '02: The Sixteenth Systems Administration Conference.

Philadelphia, birthplace of the U. S. Constitution, seems an appropriate venue for the one conference that defines and discusses the evolving constitution of the profession of system administration. LISA remains a unique and essential conference created for system administrators by system administrators, a place where practicing system administrators, researchers, software developers, and vendors meet to explore the problems of the day, learn new solutions, and predict future research challenges and solutions.

Strongly driven by social and economic forces, the constitution of our profession is rapidly changing. The informed professional must commit to continually learning new skills to keep pace in this rapidly evolving Internet culture. Every system administrator must now understand the fundamentals of networking and security, so LISA has been expanded to cover these and other timely topics that now are essential knowledge for all system administrators.

Since its inception, LISA has been an ongoing testament to the problems of the day and challenges of the future. The unforgettable events of September 11, 2001, added some momentum to the already increasing importance of recovery planning, network security, and service monitoring. Several papers in this Proceedings ask some very interesting questions for the future: Is exact replay of a journal of changes the only way to reliably reconstruct the exact behavior of a host? Is it possible to take security too far and "break the Internet"? Is the optimal time to apply a security patch immediately after release, or should one wait up to two weeks? Is there a simple mathematical model that can explain the cost of downtime and lack of redundancy to managers? The answers to these and other questions are sure to create controversy. Finding the answers, though, is very important to the future of the profession.

LISA continues the tradition of providing an integrated experience for attendees that is much more valuable than the sum of its parts. Along with you, I will be taking advantage of the many diverse opportunities LISA offers for professional growth: Sharpen your skills with tutorials on upcoming technologies. Gain insight into the nature of the profession in Invited Talk sessions. Evaluate new approaches to automation, monitoring, security, and the evolving theory of system administration (among other topics) in the Refereed Paper sessions. Find people with similar interests or administrative problems at "Birds-of-a-Feather" sessions. Bring your perplexing technical questions to experts at LISA's "The Guru Is In" sessions. Explore the latest commercial innovations at the Vendor Exhibition. Explore research directions in the accompanying advanced Workshops. Explore the controversial issues of the day with peers, researchers, and the people behind the Internet in the various social events and the famous "Hallway Track."

Thank you for your interest in LISA and, on behalf of all the organizers, we wish you a stimulating and rewarding conference experience.

Alva Couch
Program Chair

Work-Augmented Laziness with the Los Task Request System

Thomas Stepleton – Swarthmore College Computer Society

ABSTRACT

Quotidian system administration is often characterized by the fulfillment of common user requests, especially on sites that serve a variety of needs. User creation, group management, and mail alias maintenance are just three examples of the many repetitive tasks that can crowd the sysadmin's day. Matters worsen when users neglect to provide necessary information for the job. They can grow bleakest, however, at volunteer-run or otherwise loosely-coordinated sites, where sysadmins often collectively hope for someone else to attend to the task.

The Los Task Request System addresses all three problems. It mitigates user vagueness with web forms generated from XML parameter specification files. It skirts sysadmin sloth by requiring one simple review and approval step to set changes into motion. It then saves time by automatically executing commands tailored from user input. Amidst this convenience, cryptographic signatures on Los directives ensure that only administrators can alter the system. Overall, Los aims to make life easier for users and sysadmins by standardizing and streamlining the submission, review, and execution of requests for common system tasks.

Introduction

For over a decade, the volunteer student system administrators of the Swarthmore College Computer Society (SCCS) have provided shell, mail, and web services to hundreds of College-affiliated users. However, a problem arose during the 2001-2002 school year: nobody was volunteering to take care of common system administration requests. The sysadmins had an excuse: most were seniors that year and were confronted with the double whammy of the formidable Swarthmore workload and figuring out what to do after college. Still, the requests kept piling up.

Immediately, the SCCS chose to hire new sysadmins from the freshman and sophomore classes. At the same time, however, an idea began to take form. Instead of having users mail the admins with only vague ideas of what they need to say to get things done, what if a web form could guide them in supplying the necessary information? Then, what if the sysadmins could just direct the data to some handy scripts and have everything taken care of automatically? The notion of turning the e-mail client into a system administration tool was compelling, and through an impossible feat of time management, development of the Los Task Request System began.

From the onset, it became clear that Los would have to satisfy some challenging requirements:

1. It would have to be general enough to handle many different types of system administration tasks.
2. It would have to reduce the time required for common system administration tasks beneath the threshold of the harried student volunteer.
3. It would have to collect and present necessary

system configuration information to the user in order to be user-friendly (e.g., no rote memorization of group names).

4. It would have to be secure by design. Only sysadmins should be able to make changes to the system, and integrating new tasks into Los should never compromise its security.

Happily, after months of programming, Los appears to fulfill all of these requirements. Points 1 and 3 were handled by diligent coding of no particular novelty; the approach to points 2 and 4, on the other hand, is Los's most compelling feature.

Los can be characterized as a "semi-automatic" system administration tool. Some system administration tools directly empower the user to make important changes to the system. These "fully automatic" tools are carefully written to resist malicious behavior on behalf of the user; however, since they must have elevated privileges, there's always a slight risk of an exploit. Los is designed so that only the sysadmins can activate the privileged part of the system. A review of the user's input, or "task request," by a responsible human, while relatively brief and unchallenging, is mandatory.

The best way to understand how the Los system works is to follow it as it handles a single task request. This "bird's eye view" will reveal that there are many steps involved in the process. However, it is important to remember that users and sysadmins themselves only see a small and manageable fraction of them for any given request.

The process starts on the Web, where the user makes a selection from a catalog of available automated tasks (Figure 1). This catalog is generated from a collection of *task description files*, which are XML

files that contain all the information Los needs to solicit and apply task information from a user.

Using information from the description file for the user's chosen task, the Los web interface retrieves information from the user with a "wizard"-style series of input forms (Figure 2). The description file can invoke sophisticated filters that check the validity of input and solicit corrections (Figure 3).

When the user finishes entering data, Los checks the e-mail address they specified by sending them a

verification message. The user visits a web address from the message, and Los sends their input on to the sysadmins. The user is finished and now waits for the task request to be fulfilled.

A few moments later, a sysadmin sees the task request as an XML document attached to an e-mail. Surveying the user's input, the admin decides it is valid and uses a small utility to forward the data to the Los task execution module. The utility cryptographically signs the request with the admin's GNU Privacy

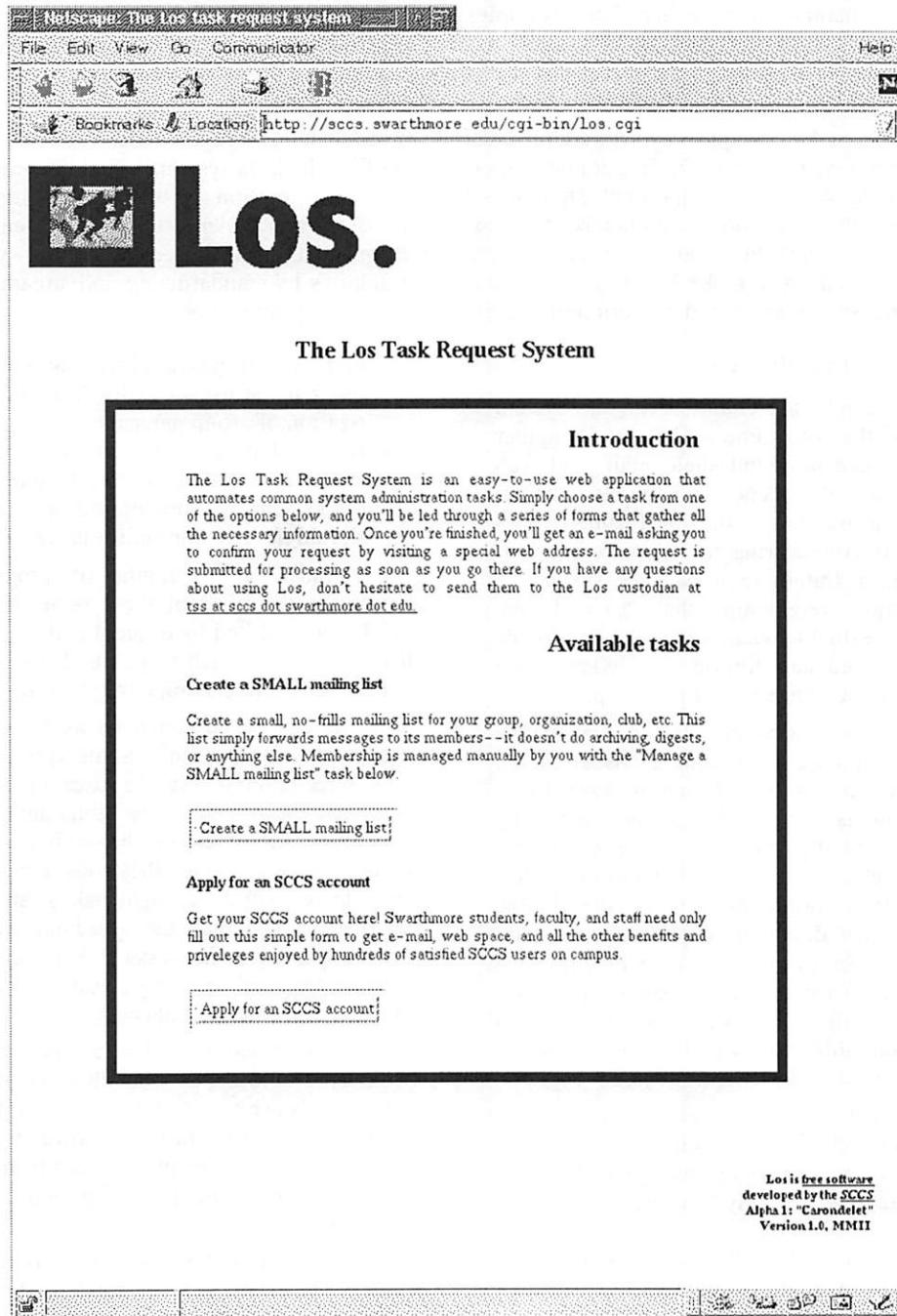


Figure 1: The Los task list. The user begins here.

Guard (GPG) [1] key before sending it. (In the future, the utility will be unnecessary; the sysadmin will simply forward a signed copy of the task request e-mail to the execution module directly.) The sysadmin is finished and waits for an e-mail confirming the execution of the task.

The Los task execution module, having validated the signature on the task request, loads the appropriate task description file and determines what commands it needs to run. It gleans arguments to the specified

commands from the task request data, and each command is executed. Finally, Los mails the commands' output to the sysadmins for review.

By now, the SCCS has successfully adapted a number of system administration tasks to this automated paradigm. Users are able to request new accounts, create and manage mail aliases and mailing lists, and allocate and control access to shared student organization webspace through six custom-made Los tasks.

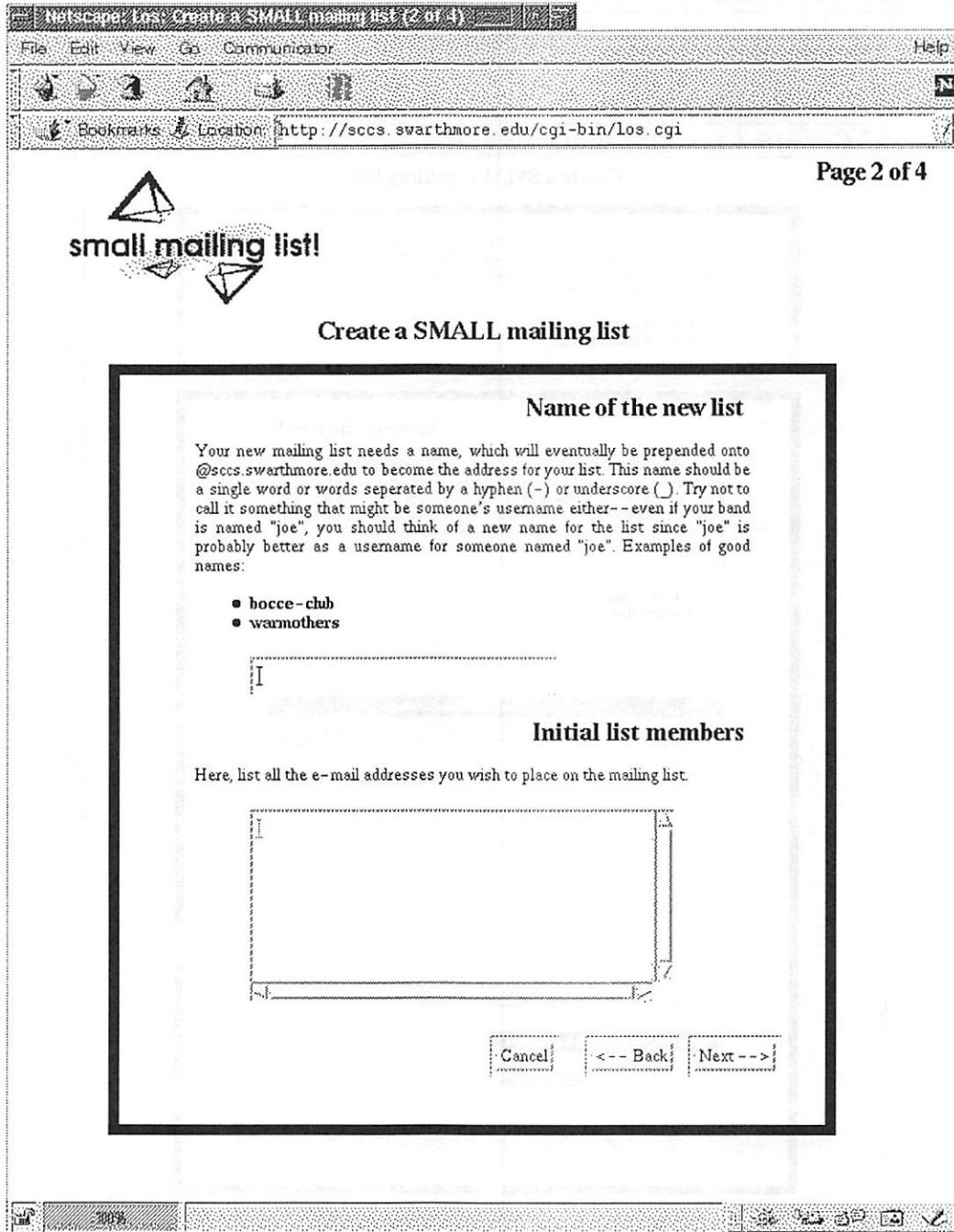


Figure 2: Using a “wizard”-style interface, the user enters data for the task.

Related work

The goals of Los are not especially novel. Several systems that automate or at least accelerate common system administration tasks already exist. These seem to fall into two categories: the fully automatic user-centric systems that require no sysadmin intervention, and systems which are intended to be seen only by the administrators. A few seem to cater to both, depending on their configuration.

A good first place to look for both kinds of software is the Internet hosting business, where the users

are owners of particular websites or other Internet resources and the administrator must oversee the servers that host them. Because site owners want to provide the same services on their sites that organizations with dedicated servers can provide, it is often necessary for system administrators to directly configure mail transport agents, FTP daemons, and other systems to their needs. Some solutions to this problem streamline the sysadmin's job: one example software package is *ispbs* [2], which provides a convenient web interface for administration of many such sites. There are several systems that also have user-centric

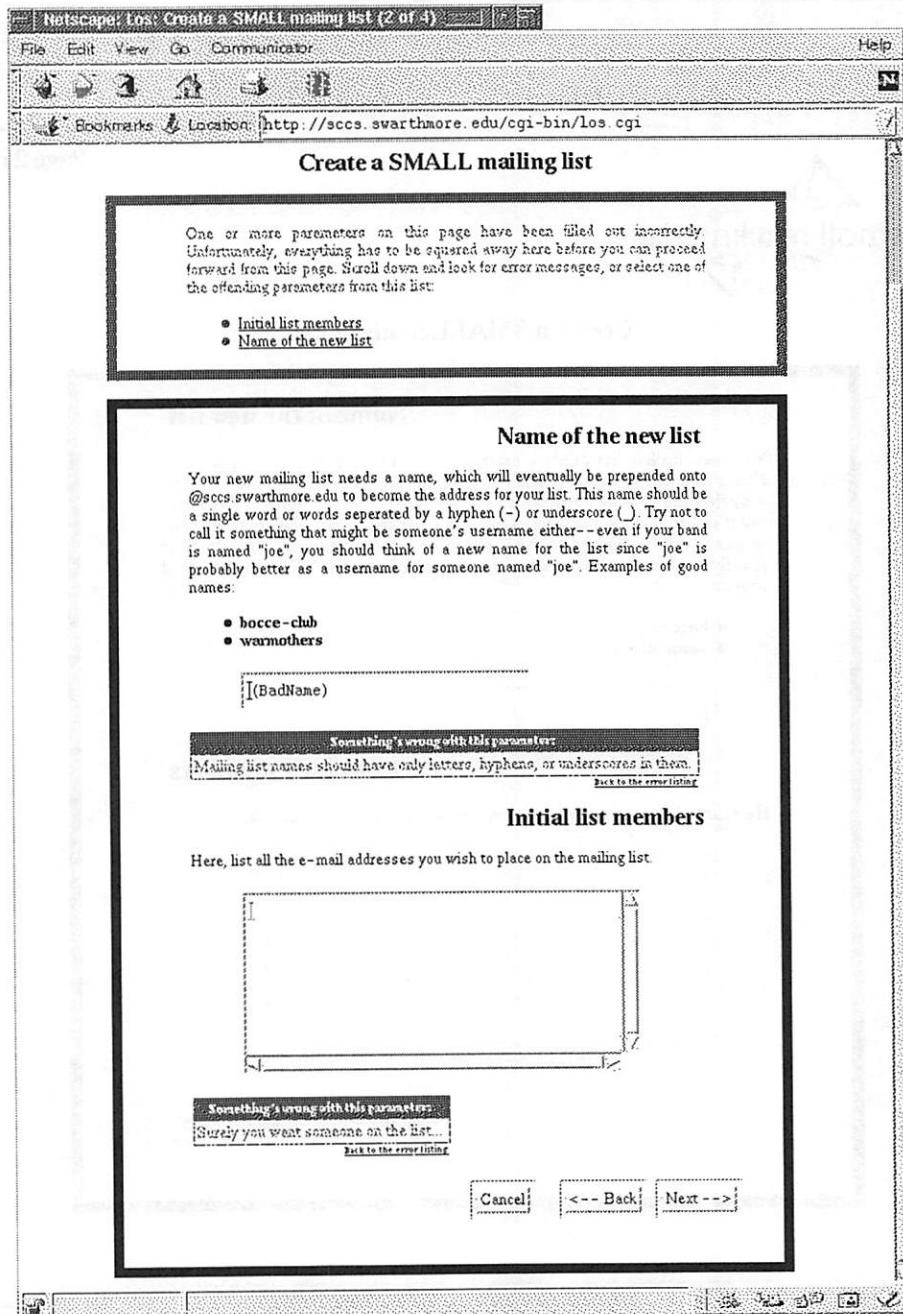


Figure 3: Task description files can invoke filters that check the validity of user input.

capabilities, however, including Account Systems Manager [3] (ASM), and ISPMAN [4] which provide web-based configuration interfaces to the user as well. To make system changes, all three eventually require some sort of automated privileged mechanism: ispbs uses a script that is automatically executed as root by cron, ASM executes changes immediately by always running as root, and ISPMAN places user requests into an LDAP database which is queried periodically by an execution system.

We find similar software outside of the Internet hosting realm as well. System administration tools that take in data from sysadmins and automatically apply it are well known and include such software as Webmin [5] and Linuxconf [6]. Both of these systems provide standardized, extensible means of gathering data from the system administrator and executing the requested changes. Linuxconf can even acquire input through different interfaces, including a web based interface, a native GUI frontend, and a text console interface. These systems also require privileges to work: like ASM, Webmin has a dedicated webserver which runs as root, while Linuxconf in common configurations uses a SUID server program executed by the xinetd Internet super server. Another systems of this sort is the Pelendor account management system [7].

Both Webmin and Linuxconf also have user accessible fully automatic capabilities. While Linuxconf does so by using a built-in per-user privilege system, Webmin employs a separate system called Usermin [8], which also features a dedicated webserver running as root. Other fully automatic systems include Accountworks [9] and Mailman [10]. These systems, which manage user accounts on a corporate network and larger mailing lists respectively, are not as general as those described above. In the case of Mailman, its narrow application focus permits it to be easily isolated from the rest of the system, thus mitigating a great deal of the security risk involved in user-activated system alteration.

The components that make up Los are also not especially new. Any user of an e-mail to FTP interface, the Majordomo [11] mailing list system, or various e-mail based problem tracking systems is familiar with sending commands by e-mail. An add-on to the RT problem tracking system [12] even checks cryptographic signatures on e-mail directives [13]. The XML encoding of user data bears a resemblance to existing XML based RPC mechanisms like SOAP [14]. Finally, the extensible web-based user input system is similar to (but rather more limited than) configurable web-based database frontends like FileMaker [15].

Los Components in Depth

The bird's eye view detailed in the introduction reveals three major stages in the life of a Los task request: creation, review, and execution. Los takes advantage of these divisions with a design that

employs a separate mechanism for each stage. While the mechanism for task request review is actually the judgment of a discriminating system administrator, the other two steps are automated by two Perl programs of considerable complexity: the web interface and task execution module mentioned previously. The web interface is a CGI program that resides in any web accessible directory that permits execution of CGI scripts. The task execution module usually resides in a library directory that contains other files necessary for both programs. The task description files, which contain detailed specifications for the data required for a task and the commands for applying it, supply both components with the specific information they need to do their job.

Both programs are designed for version 5.005 and greater of the Perl interpreter running on relatively POSIX-compliant systems. However, they also employ several different Perl modules from the Comprehensive Perl Archive Network (CPAN) [16], which may further restrict their use to the more familiar and modern Unices. A copy the GNU Privacy Guard encryption software must be installed for the Los task execution module to function. At the SCCS, Los was developed and runs on version 2.2 of Debian GNU/Linux. This section will further detail the design, use, and implementation of these important Los components.

Task Request Creation with the Web Interface

The business of getting task request information from the user is conducted by a single large CGI program that creates a series of "wizard"-like web forms for the user. While one might initially suspect that this consists mainly of presenting questions and HTML form inputs to the user, the job is much more complex for all but the simplest of data collection tasks. Much of the complexity of the Los task creation script is designed to handle the following issues:

1. **Dynamic generation of choices**. In order to keep things simple for the users, it is often necessary for the system to generate a list of possible choices on the fly rather than require the user to remember them and type them explicitly into an input box. A good example of this is a form that allows users to alter membership in a user group that has write access to a web page or other resource. It's much easier for the group members to identify their group name from a listing rather than spell it out on their own; later on, furthermore, the script must be able to list the members of the selected group for alteration.
2. **Dynamic checking of input**. Since no task request is executed without a sysadmin's approval, it's not absolutely necessary for user input to be validated at every step. However, having the system perform checks on its own can relieve the sysadmin of having to incrementally correct the user's choices again and

again. It can also allow the sysadmin to focus on more subtle errors in the input instead of typos. The issues involved in dynamic input checking are similar to those surrounding dynamic choice generation.

3. State maintenance, revision, and security.

Since the input script gathers information with a series of forms, it is necessary for it to preserve all the information the user has already supplied as it asks for more information with new forms. In the current system, this information is stored on the client side with “hidden” HTML form input elements. However, this requires tamper checking to make certain the user hasn’t maliciously altered any of the data stored on their end. In general, judicious management of input is necessary to ensure that data is kept intact through multiple back and forth transactions between client and server.

As mentioned earlier, the Los input script first greets the user with a catalog of available tasks. This simple task is accomplished with a cursory scan of all the task description files for title and summary information and is not especially complicated. More thorough examination of the task description files happens when the user selects a task and begins supplying data. Because the input script maintains all state on the client side, the following steps generally take place on each new page load:

1. The script organizes information it loads from the current task description file.
2. It determines which page of the wizard-style input forms the user has just completed.
3. If the user is advancing to the next page, it checks their new input against the specifications in the task description file. If there is a problem with the input, it prepares to show the last page again with error messages; otherwise, it readies the next screen. If, on the other hand, the user chose to go back to a previous page,

the script prepares to go backwards without checking input.

4. The script now displays the new page for the user, a process consisting of generating the HTML form elements that belong on the page and storing all of the data the user has entered so far. Data entered on other pages are cached in hidden HTML form inputs – the rest, if the user has been here before, is stored in the actual form elements shown on the page. The style of the page itself is determined by a collection of templates that can be configured by the administrator.

Each page generated by the input script is described by one of several parameters sections of the task description file. The parameters section consists of multiple parameter entries, which each describe a particular piece of information needed for the task. In addition to providing a short description of the parameter for the user, these entries also specify the proper type of HTML form widget for acquiring the information and a list of tests that validate the user’s input. Some parameters, like the user’s e-mail address, are required for all tasks; currently, if a task description file omits them, the input script will automatically insert default stand-in parameter entries into its own in-memory representation of the task.

Figure 4 contains a sample parameter entry from a Los task description file. The selector tag invokes a routine in a “standard library” of HTML form elements to generate the simple text input widget needed for this parameter. The following format tags are either Perl-compatible regular expressions (PCREs) or library calls like the selector tag, identified with pcre and filter tags respectively. For specialized applications, admins may create their own collections of selectors and filters if they choose.

Selectors and filters are nothing more than Perl routines, and relatively little effort is made to shield

```
<parameter name="uname" title="Preferred username">
  <description>
    Please choose a new username here. Be sure to specify one that is at
    least three letters long.
  </description>
  <selector name="Los::Selectors::Input" args="size=8,maxlength=8"/>
  <format>
    <pcre>/.../</pcre>
    <description>This username is too short</description>
  </format>
  <format>
    <filter name="Los::Filters::Tolower"/>
    <description></description>
  </format>
  <format inverse="true">
    <filter name="Los::Filters::IsUser"/>
    <description>This username is already taken</description>
  </format>
</parameter>
```

Figure 4: An simplified example of a parameter entry from a Los task description file.

them from the internals of Los. This means that they can actually alter the data submitted by users, a desirable feature in some cases. Because the example parameter entry in Figure 4 is soliciting a username from a new user, it uses a filter called Los::Filters::Tolower to convert the input to lowercase characters. This is necessary for the next filter, which checks whether the username already exists on the system. Some elaborate filters take even more advantage of this freedom: for example, a system of password filters for protecting access to certain Los scripts checks and updates a password database as it processes user input.

Many arguments to selectors and filters can be interpolated, thereby incorporating user input into their operation. This is the basis of the dynamic generation of choice mentioned earlier. Below, an example selector tag generates a textarea for editing group membership:

```
<selector
  name="Los::Selectors::GroupTextarea"
  args="rows=10,cols=10,wrap=off,
  group=~`pagename`"/>
```

The name of the group to be edited is stored in the variable pagename, which is named between two tilde characters in the group argument to this selector. Presumably pagename was supplied by the user in an earlier form page; in the script this example was taken from, the user chooses *pagename* from a menu of student organization webpages.

Eventually the script runs out of form pages to show the user; at this point the user has entered all the information mandated by the task description file. The script does one final check of all the user input by checking every format item of every parameter. If all is well, the script creates a file storing the information as a formal Los task request, giving it a unique ID in the process. It e-mails a confirmation URL containing

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE los_transaction SYSTEM "los_transaction.dtd">

<los_transaction date="Sat, 06 Jul 2002 17:54:49 EST"
  ip="24.205.87.222"
  id="1025996089-489825418">

<taskinfo title="Create a BIG mailing list"
  version="1.0"
  creator="tss"
  taskfile="maillist_big_new.xml"
  md5sum="d64d1c85c8c7f3700826d4eda32d512f" />

<parameters>
  <parameter name="EMAIL">tss@scs.swarthmore.edu</parameter>
  <parameter name="password_check">1a@S3d$F</parameter>
  <parameter name="pledge">null</parameter>
  <parameter name="password">1a@S3d$F</parameter>
  <parameter name="FULLNAME">Tom Stepleton</parameter>
  <parameter name="DESCRIPTION">null</parameter>
  <parameter name="listname">test-list</parameter>
</parameters>
</los_transaction>
```

Figure 5: A sample Los task request – this one requests a new mailing list named test-list.

the ID to the user and keeps the file on hand until the URL is visited. Once this occurs, the task request is sent at last to the system administrators for review and approval.

A great deal of effort has gone into making the Los input script flexible enough to handle many different types of information gathering applications. In recent months, the standard selector and filter libraries have grown in their abilities to draw information from files, user and group databases, and other sources required by the Los tasks designed for the SCCS. Nevertheless, it is certain that there will be applications at other sites where these routines will be inadequate. Even the input script itself is limited to proceeding linearly through the lists of parameters in the task description files – although it can modify its questions based on prior user input, it cannot adopt radically different branches of questioning on any basis. Thankfully, because Los's modular design permits other interfaces to create task requests to do what the web interface does, tasks with complex data gathering needs can be handled by custom applications and still work with the rest of the Los system.

Task Request Execution by Mail

Figure 5 shows a complete Los task request, which the system administrators receive as e-mail attachments from the input script. If the request is satisfactory, the sysadmin executes the task request by signing it with their GPG key and sending it on to the task execution script over e-mail. This relatively simple gesture belies the considerable complexity involved in making certain that the task request is trustworthy and avoiding the perils that come with an e-mail activated system with elevated privileges.

Because the formatting of e-mail messages differs between mailers and because the Los executor is designed to diminish security risks by analyzing all aspects of an input e-mail, a special utility program is

currently required to generate e-mails for the executor. The sysadmin pipes the task request into the utility, which encodes it in Base64 to preserve its formatting, solicits the admin's GPG passphrase, signs the data with the admin's GPG key, and sends it on to the executor.

The Los executor is currently a SUID root Perl script. This should rouse concern in cautious system administrators, as SUID scripts are widely reputed to be dangerous [17]. However, the Perl interpreter on Unix systems has a special facility for safer execution of SUID scripts: suidperl, which, by automatically imposing a technique known as *taint checking*, requires the programmer to properly shield the actions of the SUID script from externally controlled influences like input and environment [18]. There are further security precautions taken by suidperl to thwart the subversion methods commonly directed at SUID scripts, and modern Unix systems often have mechanisms that prevent the race condition attacks that made all SUID scripts unsafe in past years. Still, some admins may be unwilling to adopt the extra risk that accompanies a new program with root privileges and may conclude that Los is not appropriate for their sites.

When a signed task request is sent to the Los executor by mail, it is actually sent to a dedicated user whose e-mail is redirected by Procmail [19] or an analogous mechanism into the Los execution script. The script itself is owned by root and is otherwise exclusively executable by members of a dedicated group, of which the Los executor user is the only member. Security conscious sysadmins may elect to impose additional constraints of their own on the mechanism that directs e-mail into the Los execution script, such as an independent verification of the cryptographic signature on the task request or a blanket rejection of all messages except those from a few select hosts. Thus, though e-mail may seem like a particularly unprotected mode of transit for the task requests, judicious application of common e-mail utilities like Procmail make it possible to carefully inspect and filter them first.

Once underway, the Los executor first checks the GPG signature on the message. To do this, as it does with any external program call that doesn't require root privileges, the executor spawns a subprocess that adopts the UID of the dedicated Los user, performs the work itself, and reports back to the parent through UNIX pipes. For this particular step, a double function is served, as the Los user also owns the GPG keyring against which the task request signature is validated. For the signature to be approved, the signer's public key must be a trusted key (i.e., it must be locally signed with the Los user's key, requiring the admin to su to the losuser and import and sign their key manually) and must be explicitly mentioned in a file containing a list of keys whose owners have authorization to approve task requests. Unless all of these conditions are met, the Los executor will abort and report the failure to the sysadmins.

The script moves on to carefully decode and parse the Base64 encoded task request, maintaining a healthy paranoia about unexpected input. This done, the script examines the taskinfo tag from the task request to determine which task description file was used to generate it. The title, version, and creator attributes must correspond exactly with the version and creator information specified in the task description file, otherwise the script assumes that two different versions of the file are in use (a situation that might arise if the input script and executor are on different hosts) and aborts. Optionally, the executor can compare the MD5 checksum of its copy of the task file with that of the one used by the input script. This is not enabled by default, however, as some admins may choose to have different task description files for task request creation and execution.

With all its suspicions allayed, the Los executor can turn its attentions at last toward executing the task. There is still one contingency to anticipate, however. It is possible that the same task request might be sent to the executor twice by two sysadmins acting independently. For certain tasks, this could be harmful to the system. The executor prevents this by attempting to deposit the contents of the task request into a file named with the task request's ID in a designated log directory. If the file already exists or if the script is unable to get an exclusive lock on an empty file, the executor presumes that the task has already been executed and aborts. This protective feature also doubles as a convenient logging mechanism.

The Los executor finally spawns a subprocess to execute the task. The commands for task execution appear in a commands block at the end of the task description file, which also specifies the username under which the commands should be run. Immediately the subprocess drops as many privileges as it can and executes each command one by one. The Los executor has a relatively flexible means of interpolating variable names in command arguments. However, it is not as flexible as the shell and is not intended to be. Rather than rely on a sophisticated command line interpreter built into the script, it is expected that administrators will simply pass variables into shell scripts that do most of the work themselves.

Once the subprocess is finished, its output and the output of all the commands it invoked are sent back to the sysadmins. The well-traveled and highly-automated life of the task request is over, and the user is (hopefully) satisfied.

Los In Use at the SCCS

The Swarthmore College Computer Society has prepared a number of tasks for automation by Los. These tasks represent a certain critical intersection between those that are most frequently requested by our users and those that are the most bothersome to take care of. These include the creation of new users

and mailing lists, the management of mail aliases, and the creation and management of student organization webpages. These are interesting problems as they require both the input and execution sides of Los to involve themselves deeply in the analysis and modification of different aspects of system configuration.

The Los task description files at the SCCS have been written to thoroughly screen user input for correctness. In cases like mail alias creation, this requires the input script to check whether the new alias name isn't already being used by users, existing mail aliases, or Mailman mailing lists on our system. Organization web page management requires the culling of group membership information as well as password-restricted access. The standard selector and filter libraries handle these jobs capably, though as new needs for Los arise, there will doubtless be a need for more library functionality.

For most of the SCCS tasks, the Los executor simply invokes an external script with the data collected from the user. Often these scripts must modify configuration files like the mail aliases database, a task which requires care even when performed by a human administrator. To make this modification task simpler, Los comes with a utility that allows the scripts to perform the modifications in a "record-oriented" manner: within a section of the file delineated by special comments, the utility adds, alters, and removes comment-delimited records of text provided by the scripts.

This utility exhibits a high degree of caution and will fail if the record section and record delimiters are not all well-formed. Similarly, whenever possible, the scripts employ the system file modification tools supplied with our Linux distribution, including useradd and gpasswd for modification of the user and group databases respectively. The SCCS believes that standardized tools are the key to safe automated modification of important system configuration files, and we abide by this in our executor scripts as often as possible.

Creating Los task description files really is programming, and the time it takes depends on how thoroughly the sysadmin wishes to check user input and how difficult it is to safely automate the execution of task requests. The selector and filter lines in parameter entries are comparable to function calls and tend to each take several different arguments. It would not be difficult to make an interface for task description file creation that uses a web browser and dialog boxes to simplify the task; indeed, this might greatly speed the process, as much of the development effort goes into manually creating the XML and remembering the arguments to selectors and filters.

For the moment, setting up Los for a new task can take some time, on the order of several hours for us at the SCCS. Furthermore, if the task requires a custom selector or filter, some rather involved Perl

programming may be required, as the interface the Los input script uses to invoke the selectors and filters is complex. This may change in the future, but current efforts are focusing on making the standard libraries more versatile and complete.

For the time being, expeditious programming of Los task description files requires planning beforehand. The admin can work backwards, starting with figuring out how tasks can be executed automatically and then determining exactly how to obtain the information needed for task execution through Los. Once choices about parameter inputs have been made, the admin can start thinking about what checks they wish to apply to the user's input. At last, with all of these things established, the admin can code up the task description file parameter entry by parameter entry. Thankfully, once the task description file has been completed, installing it into the catalog of Los tasks is as simple as dropping it into the same directory as all the other Los tasks. The task appears in the listing, ready to use, the next time the main Los catalog page is loaded.

Los was completed at the SCCS in the final months of the 2001-2002 school year. As such, most students at Swarthmore were too busy to request the tasks Los has been configured to handle. After the end of the semester and up to the time of writing (mid summer), requests have been understandably sporadic. Los has indeed capably handled these requests and has dramatically improved sysadmin response time, usually finishing within a couple hours of the request submission the business that could take up to a week depending on the demands of our courses or the distractions of summer.

However, it has yet to face the normal SCCS request workload or the heavy period that comes at the beginning of the school year. The SCCS fully expects Los to greatly improve our service to the college community under these stresses, and by the time of the 2002 LISA conference we intend to quantitatively demonstrate this improvement.

One thing we are capable of measuring now is the performance of the Los system on the SCCS servers. Currently we're running both the Los input script and the executor on our main login server, a 400 MHz Pentium II-based machine with 380 MB of memory. Because Los is frequently opening files, generating NIS or LDAP queries, or doing whatever it needs to do to get the information for its selectors and filters, it is not a champion of speed.

The Los task catalog on the SCCS takes just under four seconds to load on the Swarthmore network, with the time mostly occupied by the superficial scan of the six different task description files we use. For some of the more complicated tasks, it can take about the same time to proceed from one wizard screen to the next. Even when the input script is just

creating HTML without doing any input checking or complicated widget generation, the overhead of loading and parsing the task description file and otherwise getting things ready can take about a second.

Naturally, the speed of the execution script depends on the particulars of the task being executed. Though the SCCS task description files exhibit considerable complexity when it comes to checking the user's input, the fact is that once the input is collected, there isn't much work to do for our tasks. Typically a few files will be modified and group membership will be altered, and then the task is finished. Our non-scientific gauge of how long a task request takes to execute, which involves approving a task and then enthusiastically mashing the TAB key in the Pine mailer's message index, indicates that most of our tasks take between five and ten seconds to be executed.

Future Directions in Los

After months of development, Los has grown into a system that meets all the goals the SCCS set out for it. It provides a straightforward interface for common system tasks, eliminating the usual e-mail dialog needed to determine exactly what the user wants. It provides an antidote to the "someone else will do it" syndrome of the busy volunteer sysadmin by drastically reducing the time it takes to attend to these tasks. However, Los is surely not right yet for everyone. This section lists some possible improvements to Los or similar semi-automatic system administration systems.

More thorough task delegation. For the SCCS, Los abbreviates the time it takes to attend to system tasks enough that it's not necessary to formally assign task requests to system administrators to ensure that someone attends to them. Indeed, this is probably not a good strategy for us, as it's hard to predict when a particular admin has time to attend to the system rather than coursework. However, it might make sense at some sites for task requests to be automatically delegated to members of the administration team. Modifying Los to send task requests to particular administrators would be fairly easy – making a system that carefully manages who was assigned what would be harder.

Accountability through cryptography. At the moment, Los doesn't record who authorized the execution of a task request. Since a cryptographic signature is required for this to happen, a great deal more could be done to indicate incontrovertibly who authorized the execution of a task. This also would demand relatively little modification of Los, though it does demand a secure, external means of logging task requests and signatures.

Use of XML based RPC standards. As hinted in the references section, a Los task request is little more than a remote procedure call. Los happens to use XML for task requests and task description files

mostly due to the great amount of support for XML in Perl and elsewhere, and thus relatively little attention was given to modeling Los's data transaction formats after established XML-based standards. However, it may be more beneficial from an integration and versatility standpoint to use one of the standard XML-based RPC message formats such as SOAP [14] or XML-RPC [20]. Los task requests could then conceivably be used with other systems besides the Los executor.

Integration with Linuxconf. As mentioned previously, Linuxconf is a powerful collection of tools designed to automate and provide a straightforward interface for common system administration tasks. Unlike Los, however, Linuxconf is designed for the system administrator and focuses on the kinds of system parameters the user shouldn't necessarily have to deal with (firewall setup, printer configuration, etc.).

At sites with a lot of personal Linux workstations, however, users may legitimately wish to alter these system parameters of desktop machines while administrators might prefer not to give them root access. One solution might be to use Los to collect configuration requests from the user and then to invoke the powerful Linuxconf modules with the Los executor to make the changes.

Linuxconf already does much of what Los does with respect to collecting and applying data, so it may instead make sense to adapt Linuxconf to the semi-automatic approach to task request approval.

Easier review of task requests. Right now the review of Los task requests requires the sysadmin to visually parse XML to determine whether the user's input is appropriate. This is not especially difficult, but it could be streamlined by a program that interpreted Los task requests, combined them with the information in their corresponding task description files, and generated more legible representations of the user's data. Standardized technologies like XSLT [21] could make this a rather straightforward task,

Easier authorization of task requests. One extremely desirable improvement to Los is the elimination of the clumsy approval script. It would be better if the executor were capable of taking signed e-mail forwards from any mail client, determining whether the signature was valid, and executing the task request. This is difficult, however, as it requires careful analysis of the e-mail, and of course the consequences of a misjudgment could be dire. Further complicating matters, some mailers have different behaviors when it comes to signing e-mails with attachments, let alone signing forwards of e-mails with attachments. Still, the benefits of being able to simply forward a task request to the Los executor make this an eminently worthwhile goal.

Novel input methods. The web interface to Los is a fairly satisfactory means of acquiring input from the user. Pains have been taken to make the default

template for Los's HTML output attractive in Lynx and other text-based browsers. Still, it might be useful to be able to submit Los task requests from handheld computers, kiosks, embedded specialty systems, or other devices where web browsers are not practical. Unfortunately, this may require a restructuring of Los, as the selector and filter lines in the task request files are tied fairly exclusively to the Web-only standard libraries.

Integration with problem tracking systems. Though not necessary for the SCCS, some sites might benefit from managing Los task requests with a problem-tracking system. Users could check on the status of their task requests, and sysadmins could tell at a glance which tasks were awaiting attention. A history of executed tasks would also be available for later perusal.

Limited experimentation on integrating Los with problem tracking software has already taken place. Because Los uses e-mail as its transaction transport mechanism, the e-mail based GNATS system [22] was an easy choice. It was not difficult to alter the CGI input script and the task request approval script to create and interpret GNATS problem report e-mails. However, there are opportunities for tighter integration. Just as you can now edit a problem report by specifying its category and ID number on the command line (as in `edit-pr mycategory 532`, a sysadmin should be able to approve a task request in the same fashion with a script that automatically updates the status of the problem report that contains it. This should not be a challenging task.

One issue that remains to be resolved is the encoding of task requests in GNATS problem reports. When the input script sends a task request to the sysadmins, it places it in a Base64 encoded MIME attachment to avoid corruption of the data. Most Los applications don't need this precaution, but the risks of quoted-printable mail encoding, CR to CRLF conversion, and other e-mail mutations make it a prudent one. A default GNATS installation does not deal well with MIME-encoded e-mails, though this issue is being addressed. For the time being, though, another encapsulation mechanism may be necessary.

Use on multiple systems. At the moment, thanks to the stopgap task approval script, Los task requests can only be directed toward a single Los executor, and thus a single computer system. Certainly the script called by the executor could use a tool like Igor [23] to trigger changes on many systems at once. What about a situation where different kinds of task request must be executed on different machines?

When the task approval script is eliminated, the sysadmin will be able to simply direct task requests toward the proper computer by altering the To: field in their e-mail client. However, because executing a task request on the wrong system could have negative consequences, it might be appropriate to also place extra information in Los task requests that prevent them

from being executed on the wrong system. This would not be an especially difficult modification.

Getting Los

In order to encourage widespread use and enthusiastic development, Los has been released under the most recent version of the BSD license. Los can be downloaded from the Free Software Foundation's Savannah development repository at <http://savannah.gnu.org/projects/los/>.

Acknowledgments

Deserving recognition are the Swarthmore College Computer Society and the Swarthmore Linux User's Group mailing list members for their resources, support, and encouragement. Advice and instruction from the denizens of #perl on irc.openprojects.net as well as from Frank J. Tobin, the author of the GnuPG::Interface Perl module, are gratefully acknowledged.

Author Information

After getting his start administering the Linux mail and web server at tiny Thomas Jefferson School in St. Louis, Tom Stepleton went on to serve as a sysadmin at the Swarthmore College Computer Society for all four of his undergraduate years. Currently, Tom is a first year doctoral student at the Carnegie Mellon University Robotics Institute, where some systems can physically evade administration. While Tom tends to crank out open source software packages semiannually, Los is his largest to date.

References

- [1] GnuPG Team, et al., "GNU Privacy Guard," <http://www.gnupg.org/>.
- [2] Host Plus, et al., "ispbs," <http://ispbs.hostplus.net/>.
- [3] Neikous Software, et al., "Account Systems Manager," <http://asm.neikous.com/>.
- [4] Ghaffar, Atif, et al., "ISPMan," <http://www.ispman.org/>.
- [5] Cameron, Jamie, et al., "Webmin," <http://www.webmin.com/>.
- [6] Solucorp, et al., "Linuxconf," <http://www.solucorp.qc.ca/linuxconf/>.
- [7] Curtin, Matt, Sandy Farrar, and Tami King, "Pelendur: Steward of the Sysadmin," *Proceedings of the Fourteenth Usenix System Administration Conference*, Dec. 2000.
- [8] Cameron, Jamie, et al., "Webmin," <http://www.webmin.com/index6.html>.
- [9] Arnold, Bob, "Users Create Accounts on SQL, Notes, NT, and UNIX," *Proceedings of the Twelfth Usenix Systems Administration Conference*, Dec. 1998.
- [10] Viega, John, Barry Warsaw, and Ken Manheimer, "Mailman: The GNU Mailing List Manager."

- [10] *Proceedings of the Twelfth Usenix Systems Administration Conference*, Dec. 1998.
- [11] Chapman, D. Brent, "Majordomo: How I Manage 17 Mailing Lists Without Answering "-request" Mail." *Systems Administration (LISA VI) Conference (LISA '98)*, Oct. 1992.
- [12] Vincent, Jesse, et al., "RT: Request Tracker," <http://www.fsck.com/projects/rt/>.
- [13] Vincent, Jesse, "enhanced-mailgate," <http://www.fsck.com/pub/rt/contrib/2.0/rt-addons/enhanced-mailgate README>.
- [14] W3C, "W3C Recommendation: Simple Object Access Protocol (SOAP) 1.1," <http://www.w3.org/TR/SOAP/>.
- [15] FileMaker, Inc., "FileMaker: Products: FileMaker Pro 6," <http://www.filemaker.com/products/fm-home.html>.
- [16] *The Comprehensive Perl Archive Network*, <http://www.cpan.org/>.
- [17] Akin, Thomas, "Dangers of SUID Shell Scripts." *Sys Admin Magazine*, June 2001.
- [18] Birznieks, Gunther, "CGI/Perl Taint Mode FAQ," <http://gunther.web66.com/FAQS/taintmode.html>.
- [19] van den Berg, Stephen R., Philip Guenther, et al., "Procmail," <http://www.procmail.org/>.
- [20] UserLand Software, et al., "XML-RPC," <http://www.xml-rpc.org/>.
- [21] W3C, "W3C Recommendation: XSL Transformations (XSLT) Version 1.0," <http://www.w3.org/TR/xslt/>.
- [22] Free Software Foundation, et al., "GNATS," <http://www.gnu.org/software/gnats/>.
- [23] Pierce, Clinton, "The Igor System Administration Tool," *Proceedings of the Tenth Usenix System Administration Conference*, Sept. 1996.

Spam Blocking with a Dynamically Updated Firewall Ruleset

Deeann M. M. Mikula, Chris Tracy, and Mike Holling – Telerama Public Access Internet

ABSTRACT

In this paper, we detail our methods for controlling spam at a small ISP, reducing both resource usage and customer complaints. We will discuss our initial unsuccessful tactics, and the resulting development of our unique spam blocking system. Deny-Spammers classifies hosts as probable spammers and inserts those hosts into a dynamically updated firewall ruleset on our mail server, thereby effectively blocking the host from making an SMTP connection to our mail server. Our analysis demonstrates that this has been effective in reducing the amount of spam that our customers receive, and the burden on our limited resources.

Introduction

Are you aggravated by spammers launching what are effectively Denial of Service Attacks against your mail server? We were, and after several attempts at using some established spam-control techniques, we recognized the need to create our own novel approach, which we affectionally call “Deny-Spammers.”

With the abundance of spam on the Internet today, nearly every ISP finds themselves forced to participate in some kind of spam¹ blocking.

Jon Postel wrote in RFC 706 [1], “there is no mechanism for the [email] Host to selectively refuse messages. This means that a Host which desires to receive some particular messages must read all messages addressed to it. Such a Host could be sent many messages by a malfunctioning Host. This would constitute a denial of service to the normal users of this Host. Both the local users and the network communication could suffer.”

While this scenario is common enough today, it was a shocking thought in 1975 when Postel authored RFC 706. Then, the Internet was still a place of co-operation where users operated with the “greater good of the net” in mind. Today’s mail servers operate under an almost constant threat of Spam Denial of Service attacks.

In an article in The New York Times from June 27, 2002, Jennifer Lee writes, “Brightmail, which maintains a network of In boxes to attract spam, now records 140,000 spam attacks a day, each potentially involving thousands of messages, if not millions.” [2] A similarly bleak report from Hotmail states that 80% of its almost two billion processed email messages are spam [3].

Of particular interest to us as an ISP is the reaction of the customer base to spam in their inbox. A

¹“Spam” is the term commonly used to refer to mass-emailed, unsolicited commercial email (also known as UCE), sent by a person or organization usually referred to as a “spammer.”

report by Gartner Consulting states that 53% of its respondents place the blame for spam on their ISP. They found that UCE (Unsolicited Commercial Email) ranks fourth in reasons for customer churn [4].

For these reasons, having a spam control policy is no longer an option for an ISP, no matter what its size. Hotmail subscribes to MAPS (Mail Abuse Prevention System) RBL (Realtime Blackhole List) [5]. Both AOL and Earthlink advertise their spam filtering as a benefit to their services.

Telerama is a small ISP, established in Pittsburgh, PA in 1991. Our mail system consists of a single server for both incoming and outgoing mail. It uses a 1 GHz Athlon processor and has 640 MB of RAM, running FreeBSD 2.2.8-STABLE. Our mail transport agent is *qmail-1.03* [6]. In addition to the stock *qmail* distribution, we are using the *qmail-uce checklocal* patch [7] to reject mail for non-existent mailboxes.

This server handles all incoming and outgoing mail for approximately 7,000 accounts, including well over 600 hosted virtual domains, and their associated addresses. The server typically delivers between 50,000 and 70,000 incoming e-mail messages to local users in a 24-hour period. Attempted deliveries, including messages to non-existent mailboxes, varies between 100,000 to 140,000 messages in the same time period. We approximate that 50% of the attempted deliveries are blocked by the *qmail-uce checklocal* patch alone.

From the user’s perspective, spammers cause a general slowing down of our entire mail system. At times, a single spammer would open hundreds of simultaneous SMTP connections to our mail server, dumping thousands of messages into our mail queue. This causes delays in message delivery lasting from minutes to hours. A user sending mail through our system could wait up to 30 seconds before a 220 response [8] code is returned by the SMTP server. As most spam is generated during business hours, the mail queue would

shrink back to manageable sizes in the evening. The next day, the problem repeats itself. When the snappy performance we are used to diminishes, our users start to complain about the sluggish performance. We wanted to be able to identify spammers and simultaneously block them in order to prevent degradation of the performance of the mail server.

What We Tried First

Two alternate approaches we attempted before developing Deny-Spammers were:

- using *qmail-uce*'s checklocal patch to deny mail for non-existent mailboxes
- using *ucspi-tcp*'s *rlsmtpd* [9] in conjunction with several RBL sources

First, we attempted to implement the *qmail-uce* checklocal patch alone. This patch makes *qmail* reject mail for non-existent mailboxes. *qmail*, by default, accepts mail for non-existent addresses. It determines later whether or not the user exists.

Unfortunately, this did not prevent spammers from getting connected to the mail server and getting messages into the mail queue. Many spammers make several parallel SMTP connections, so it was not uncommon to see 50 or more SMTP connections from a single IP address. Although the checklocal patch prevented messages to non-existent addresses from entering our mail queue, spammers essentially created a Denial of Service to the users of our mail server.

We immediately realized that the *qmail-uce* checklocal patch would not solve our problem on its own. Our next approach involved using *rlsmtpd*, which is part of the *ucspi-tcp* [10] package that includes *tcpserver* [11]. *rlsmtpd* attempts to block mail from RBL-listed sites by querying one or more RBL sources. *rlsmtpd* works with any *smtpd* server that runs under *tcpserver*. It can be configured to respond with a permanent (553) failure error message or a temporary (451) failure error message.

We attempted to implement *rlsmtpd* in several ways. First, we configured *rlsmtpd* to run continuously and deliver a temporary (451) error to RBL-listed sites. This prompted many customer complaints regarding legitimate mail not getting through. Next, we set up *rlsmtpd* to deliver a permanent (553) error to those RBL-listed sites. Again, this caused many of our customers to complain about mail bouncing.

Our last attempt involved running *rlsmtpd* only during times when we were being heavily spammed. Because it was configured to deliver a temporary error to RBL-listed sites, it was effectively useless. Spam would just queue up on the originating server until we turned off *rlsmtpd*. At that point, it was obvious that we needed something other than *rlsmtpd*. *rlsmtpd* did not help us solve the problem of our mail server getting pummeled by spammers' SMTP connections, and it brought on many complaints.

Some other alternatives to *rlsmtpd* include utilities such as Sieve [12] or SpamAssassin [13]. Unfortunately, these utilities must process each message individually. This results in a significant increase in the overall system resources required by the mail system. Compounded with the fact that spammers were already utilizing all of our server's resources, these utilities were not an option. Another option was to buy faster hardware or multiple servers. We opted for a homegrown software solution before investing in more hardware.

Design Goals

Each site is going to have its own unique problems when implementing an "out of the box" spam filter. In order to effectively implement a spam filtration system, a site needs to address these questions:

- What has not worked for us in the past?
- Do we have enough resources to allow client-side filtering options?
- Do we have the time and expertise to create our own spam blocking solutions?
- Would it be more effective to purchase faster and better hardware than to script a custom solution?
- How transparent does the spam blocking need to be to the user base?
- Are we concerned with bandwidth consumed by spam attacks?

After addressing the questions above, and ruling out failed approaches, we realized that we needed to refine our goals for a spam filtering system, and would probably need to engineer our own software-based solution based on those design goals.

Our requirements were:

- The method must conserve system resources.
- The method must reduce the amount of bandwidth consumed by spam attacks.
- The method must not add much additional overhead to mail processing.
- The method must prevent spamming sites from getting mail into the mail queue.
- The method must be manageable in a way that allows us to exempt certain hosts or networks.
- The method must keep our customers happy by minimizing the number of false positives.
- The method must be as transparent as possible to end users.

Our number one concern was to implement a solution which solved our problem without increasing the load on our mail server, as we did not desire a hardware upgrade at the time we were developing Deny-Spammers. This limitation ruled out utilizing a processor intensive spam control system, such as SpamAssassin or Sieve.

In addition to the headaches of a major mail migration, simply throwing hardware at the spam

attacks would only be a short term solution. There has been, and will likely continue to be, a practically exponential growth in the amount of spam on the Internet. In a matter of months, our new hardware could be overwhelmed by additional and more creative spam attacks.

A hardware solution also fails to rein in the problem of bandwidth consumption by spam attacks. We simply wanted to reduce the ability of spammers to consume our resources (such as bandwidth and CPU utilization) conserving them for legitimate mail.

We choose to use frequency of attempts to deliver email messages to non-existent mailboxes as our heuristic. We later felt validated in choosing this metric because it was proposed by Jon Postel in RFC 706, which states, "A Host might make use of such a facility by measuring, per source, the number of undesired messages per unit time, if the measure exceeds a threshold, then the Host could issue the 'refuse message from Host X message...'" Other metrics, such as number of concurrent SMTP connections could be used to qualify the sending SMTP server as a spammer.

Once identified as a spammer, a method is needed to block future delivery attempts from that host to our mail server. We chose the most efficient method, which is to do the filtering at the IP level, in order to put the load of the filtration process on the operating system's kernel, rather than at the application level. Filtering at the application level, such as is done by SpamAssassin and Sieve, for example, would not prevent a spam Denial of Service attack.

Implementation/Solution

Deny-Spammers is a daemon that interfaces to the mail transfer agent and the firewall ruleset control program. It uses a strategy based upon patterns that spammers produce, including attempts to send to nonexistent addresses, to dynamically update the server's ingress rules. This approach moves blocking spam away from the delivery agent to the mail server's kernel, thus conserving system resources.

We chose to implement our filtration tool in Perl to increase the speed of development. This allowed us to have a working prototype within a few days of our idea's inception. Although the Perl implementation works well in production for our system, larger sites may want to consider using a more efficient development language. This becomes even more of a necessity as more heuristic tests are introduced.

We needed a solution that would work with the software that we were already using. Therefore, the current revision of Deny-Spammers is specifically designed for use on FreeBSD systems running *qmail*.

Presently, Deny-Spammers has two major prerequisites:

- any version of FreeBSD with IP firewall support installed in the kernel

- a patched version of *qmail* that includes the *qmail-uce* checklocal patch to reject mail for nonexistent addresses

Deny-Spammers interacts with the kernel's firewall by using FreeBSD's *ipfw* [14] program to ban and un-ban hosts. *ipfw* provides the user-level control of the firewall ruleset. Using Deny-Spammers with another operating system's firewall application would require additions to the code. Support for iptables, ipchains, packetfilter or IP Filter would all be simple to add. Modifying the system calls that Deny-Spammers makes would suffice to add support for any of these packet filters.

The types of patterns produced by spammers determines how Deny-Spammers interfaces to the mail transfer agent. In our case, spammers are detected by multiple sends to nonexistent addresses. In order to detect delivery attempts to nonexistent addresses in *qmail*, the *qmail-uce* checklocal patch was required. This patch provides a modification for *qmail*'s SMTP daemon, *qmail-smtpd*, and logs details about each attempted delivery to a nonexistent mailbox, including the IP address of the host which attempted this delivery.

Although Deny-Spammers is currently dependent on the *qmail-uce* checklocal patch, it could be adapted to function with other MTAs. The exact implementation, would depend on the MTA itself. Sendmail, for example, defaults to logging delivery attempts to nonexistent addresses.

Deny-Spammers is intended to be executed at boot time and to run continuously. Many parameters can be fine-tuned from within the script. More information about these can be found in the code itself and in the implementation details that follow. To obtain the source code for Deny-Spammers, see the availability section near the end of this paper.

Because the checklocal patch logs all attempted deliveries to nonexistent addresses, Deny-Spammers monitors the system's mail log for messages emitted by the patch. These messages are parsed for the IP address of the host that attempted to make such a delivery. By defining thresholds of how many of these messages can be seen in a given time period, Deny-Spammers selectively prohibits hosts from making any more SMTP connections for a given "ban time" period.

To produce this behavior, three hash structures are used to track the state of the spam filtering system. One is used to track the times that hosts sent undeliverable messages, another for banned spammers, and another for the exception list.

The %spammer hash is a hash of lists. The keys of the hash are host IP addresses. The values of the hash are lists for each host which contains a list of timestamps. These timestamps represent times in which a host sent mail to a nonexistent address.

The %banned hash is a plain hash. The keys of the hash are the host's IP address. The values of the hash are scalars containing the timestamp in which that host was banned.

The %noban_list hash is a 4-level hash which contains the IP address exception list. This hash is organized such that the first level represents the first set of octets, the second level represents the second set of octets, and so on. The keys of this hash represent octets, with asterisks interpreted as wildcards. The values of the hash are '1' if a host or network is on the exception list.

The exception list is populated with the contents of the exception list file specified when the program starts. It is a flat ASCII text file containing one IP or network per line. An example exception list is provided with the distribution. The list is periodically re-read by the script and any necessary firewall changes happen automatically.

There are a number of variables in the beginning of the script that may need to be adjusted based on the application. The most important parameters are the number of non-deliverable message attempts during a time span that occur and the time span itself. Ten attempts during a five-minute period has worked for us. The ban time, or length of time a host stays banned, is also adjustable. Lower ban times will typically keep the size of the firewall ruleset reasonably small.

Other variables include the path to the exception list, log files, *ipfw* and the regular expression used to match incoming timestamps and IP addresses from the mail log input. With the *qmail-uce checklocal* patch, nonexistent user messages should appear in the system's mail log as shown in Listing 2.

```

While (true) {
    Match incoming lines against a regular expression for
    undeliverable messages to nonexistent addresses and parse
    timestamp and IP address.

    Skip line if host is in the exception list.

    Trim the timestamp list for this host to $MAX_SPAMMER_ENTRIES.

    Add the timestamp to the host's list contained in the spammer
    hash.

    Check how many delivery attempts to nonexistent address this host
    has made in the sampling interval, $SPAM_TIMESPAN.

    If nondeliverable messages > $SPAM_TRIGGER then filter this IP.

    If current time >= $next_refresh then
        calculate next refresh,
        reload the exception list and prune the banned hosts list
        (un-ban hosts who have been banned for $BAN_TIME)
}

```

Listing 1: Pseudocode for infinite loop.

```

Jan 1 00:00:00 mailhost smtpd: 1234567890.123456 12345: DENYMAIL:
RCPT_TO:_Filter.NoUser:_ relay unknown [123.123.123.123] FROM
<bounce@your-info.net> ADDR <abcdefg@telerama.com>

```

Listing 2: Typical nonexistent-user message.

users. Fine-tuning the tests may also be required, again depending on their needs, to achieve the desired results.

In Production

Figure 1 shows the number of attempted deliveries to non-existent mailboxes and the number of firewall rules over a four-day period. The first two days shows Deny-Spammers running. For graphing purposes we intentionally reset it three times. The last two days show what happens when Deny-Spammers is disabled.

A five minute sampling interval was used for both the number of firewall rules as well as the number of undeliverable mail attempts.

The graph starts with one firewall rule on April 25th. At this point Deny-Spammers was initialized. At this point, the firewall rules begin to increase. As the firewall rules approach 1,000, the delivery attempts average around 100-200 attempts every five minutes.

When the firewall is reset, the firewall rules go back to one, and the delivery attempts begin to increase. As the delivery attempts decrease, the firewall

ruleset starts increasing at a slower rate. Notice that the firewall rules increase very quickly when the number of delivery attempts counts is high and the firewall has just been reset.

Shortly after midnight on April 27th, the firewall is at 1 for the third time. The delivery attempts increased dramatically when Deny-Spammers was disabled for about six hours.

Deny-Spammers was restarted where the firewall rules start to increase again (on the morning of April 27th). A similar pattern occurs, the delivery attempts decrease as the firewall rules increase.

In the last two days of the graph, Deny-Spammers was completely disabled. The delivery attempts average more than twice that of when the spam filtration system was enabled.

Limitations

Despite its usefulness, Deny-Spammers has some limitations.

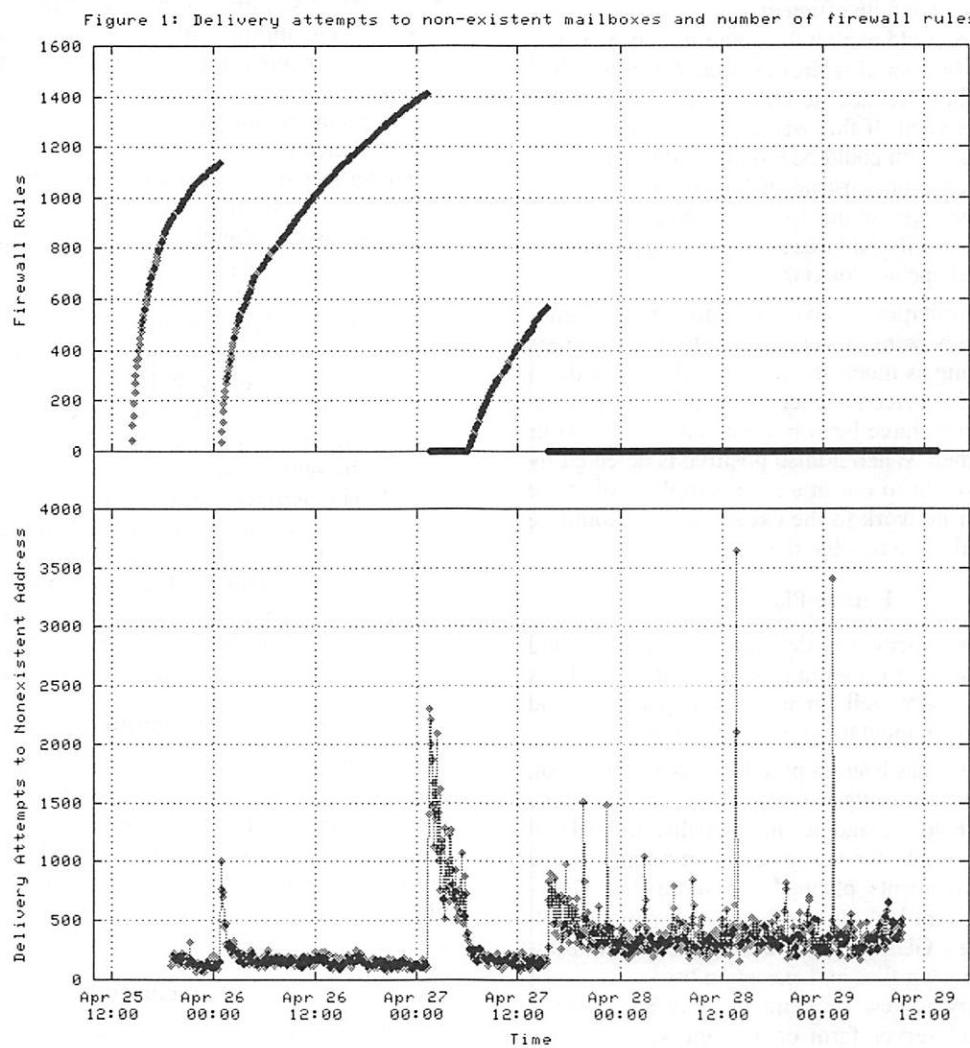


Figure 1: Attempt deliveries to non-existent mailboxes and number of firewall rules.

- The exemption list only supports individual hosts and/or classful networks. CIDR notation is not supported.
- Deny-Spammers is currently only compatible with FreeBSD machines running *qmail*.
- Another known issue with the software is that *qmail-smtpd* processes will hang for a while if a host is banned while they have an active connection. This is mostly a problem when the script is first started as the initial surge of banning causes a large number of such hung processes.
- Scalability is limited in some cases. The kernel firewall code itself may not be able to efficiently process thousands of rules, which is a common scenario. We have determined that older revisions of FreeBSD, for example, are subject to this problem. With older versions, firewall rules were stored in a linked list. Newer revisions use a tree structure, making the handling of large rulesets much more efficient. Using an inefficient data structure ultimately caps the maximum number of rules that a server can handle efficiently.
- Spammers could exploit the *qmail-uce* checklocal patch to find valid addresses. *qmail*, by default, doesn't allow a sender to know whether or not an address is valid. If this was a major concern, the checklocal patch could be modified so that it only logs the attempts, instead of logging and bouncing the message. In this case, spammers wouldn't be able to verify addresses, and Deny-Spammers would still operate correctly.

These limitations have not prevented Deny-Spammers from being a very useful tool. Customers are not receiving as much spam and it effectively deals with Denial of Service instances caused by spam. Customer complaints have been minimal compared to our other approaches. When a false positive is detected by a user and brought to our attention, simply adding the correct host or network to the exception list should be all that is required to resolve the issue.

Future Plans

Deny-Spammers was developed as a quick and dirty hack to solve a pressing problem that we had. As such, it works very well for us, on our platform, and for our staff, who understand its limitations.

Now that it has been in production for over a year, and providing the results we desire, we can see many areas in which to expand its functionality to make it attractive and usable for the general community. Some obvious improvements planned are addressing Deny-Spammer's current lack of scalability and interoperability, and adding a GUI interface to allow non-administrators to access its log files and exception list.

- Scalability issues. Implement Deny-Spammers for a mail server farm or a single server with many more users.

- Add the ability to use a separate firewall. Currently, the firewall must be located on the same machine that is processing mail. A separate firewall could be updated remotely via an ssh tunnel. This feature could also be useful when scaling this application to a mail server farm, so that one firewall could be responsible for a group of mail servers. Given a secure communication mechanism to update the firewall rules, such an improvement should be straightforward.
- Integration with third-party applications such as SpamAssassin or Anomy Sanitizer. Allow results from SpamAssassin/Anomy to determine whether or not a host gets banned.
- Improve statistical generation for research purposes. Create historical averages of number of hosted blocked over long periods of time. Look for interesting patterns, such as whether spam comes in bursts and when it most frequently occurs.
- Develop a better interface for unbanning hosts and managing the exception list. Add CIDR notation support for the exception list.
- Interoperability with other operating systems. This is simply a matter of adapting the firewall system calls to work with various firewall implementations (ipchains, iptables, IP Filter, packetfilter).
- Interoperability with other mail transfer agents (sendmail, postfix, et cetera). Because each program works slightly differently, interfacing each spam signature pattern could be a tedious process.
- Develop more "spam signatures." A few other patterns we've considered using as criteria are:
 1. The number of concurrent SMTP connections made by a host – experience has shown that spammers are capable of making many parallel SMTP connections to the same mail server.
 2. The number of recipients a message is sent to – Spammers often send messages with extremely large RCPT TO lists.
- A point system for hosts could be introduced, such that multiple spam signature patterns are taken into account for each host (similar to the 'hits' mechanism used by SpamAssassin).

Availability

Deny-Spammers is freely available source code and documentation can be found at <http://deny-spammers.telerama.com>. Deny-Spammers is written in Perl 5 and developed in and tested under FreeBSD. It contains no dependencies on non-standard modules or libraries. Specific questions regarding Deny-Spammers can be sent to denyspam@telerama.com.

Conclusions

This paper describes a stateful inspection strategy for dynamically creating firewall rules that block

access from mail hosts based upon their recent behavior. If the sending mail host is determined to be a spammer (based on our criteria) a daemon updates the firewall ruleset for our mail server.

Proving the success of our strategy is difficult due to the impossibility of measuring the lack of an event (lack of delivery of spam messages). Many alternative approaches were too resource-intensive for us to implement. We found that other spam filters that were acceptable for us resource-wise (such as MAPS-RBL) created large and obvious negative customer feedback. We have not had that backlash upon implementing Deny-Spammers. We feel that we have fewer customer complaints about receiving spam, but we don't have enough data to support that point empirically.

What our data does show is that we are banning thousands of misbehaving mail hosts based on our metrics. We believe all of these hosts to be likely spammers.

Authors

All of the authors have worked at Telerama Internet (<http://www.telerama.com>) for the past several years.

Deeann M. M. Mikula is the Director of Operations and Junior Unix System Administrator at Telerama Internet, and is co-founder of the local SAGE Chapter in Pittsburgh. She has worked as a Behavioral Neuroscience Researcher, a Coffeehouse Manager and a Visual Artist. When not in front of a keyboard, she can be found drinking scotch at a Gothic club or painting and drawing. Deeann can be reached via email at deeann@telerama.com.

Chris Tracy is Telerama Internet's Senior Network and Systems Engineer and a SCinet Volunteer. He holds a Bachelor's Degree in Computer Engineering from the University of Pittsburgh. When not in front of a keyboard, Chris can be found drinking beer, DJ'ing or playing drums. Chris can be reached via email at chris@telerama.com.

Mike Holling is a part-time Network Engineer with Telerama Internet. He holds Bachelor's Degrees in Computer Science and Electrical Engineering from Carnegie-Mellon University. When not snow boarding, skate boarding or drinking beer, Mike can be found working as a Network Consultant in Whitefish, Montana. Mike can be reached at myke@telerama.com.

Acknowledgments

We are grateful to many people for their contributions to this project and this paper. We would like to thank Doug Luce, owner and CEO of Telerama, for fostering a workplace where we are encouraged to try novel approaches to problems. We thank our fellow staff at Telerama for putting up with the real-time tweaking of our production mail server.

Especially valuable in producing this paper was peer review and support. We would like to thank

Esther Filderman and Josh Simon for encouraging us to publish our work, and for support along the way. The advice of our shepherd, John Sellens, and the comments of our anonymous reviewers, were invaluable in shaping our final paper.

References

- [1] Postel, Jon, "RFC 706: On the Junk Mail Problem," November 1975. Network Working Group. 10 April 2002, <http://www.faqs.org/rfcs/rfc706.html>.
- [2] Lee, Jennifer B., "Spam: An Escalating Attack of the Clones," *New York Times*, June 27, 2002.
- [3] Gomes, Lee, "How Hotmail Keeps Its Email Empire From Spam's Clutches," *Wall Street Journal*, July 8, 2002.
- [4] Gartner Consulting, "ISPs and Spam: The Impact of Spam on Customer Retention and Acquisition," June 14, 1999.
- [5] "Mail Abuse Prevention System LLC," <http://mail-abuse.org/rbl>.
- [6] Bernstein, Dan, "qmail home page," <http://www.qmail.org>.
- [7] Varshavchik, Sam, "MAIL 1.01 unified Anti-UCE/Mailbombing patch," <http://portofhoodsport.org/qmail/misc/uce.html>.
- [8] Postel, Jonathan B, "RFC 821: Simple Mail Transfer Protocol," August 1982. Information Sciences Institute: University of Southern California, 20 April 2002, <http://www.ietf.org/rfc/rfc0821.txt>.
- [9] Bernstein, Dan, "The rblsmtpd program," <http://cr.yp.to/ucspi-tcp/rblsmtpd.html>.
- [10] Bernstein, Dan, "ucspi-tcp home page," <http://cr.yp.to/ucspi-tcp.html>.
- [11] Bernstein, Dan, "The tcpserver program," <http://cr.yp.to/ucspi-tcp/tcpserver.html>.
- [12] Showalter, T., "Sieve: A Mail Filtering Language," January 2001. Network Working Group, 14 April 2002, <http://www.ietf.org/rfc/rfc3028.txt>.
- [13] Hughes, Craig R., "SpamAssassin home page," <http://www.spamassassin.org>.
- [14] Ugen J. S. Antsilevich, Poul-Henning Kamp, Alex Nash, Archie Cobbs, and Luigi Rizzo, "Manual page for ipfw - IP firewall and traffic shaper control program," <http://www.freebsd.org/cgi/man.cgi?query=ipfw>.

Holistic Quota Management: The Natural Path to a Better, More Efficient Quota System

Michael Gilfix – Tufts University

ABSTRACT

Disk quota systems exist to protect a limited resource and ensure that users can share it. However, existing quota management systems concentrate on controlling user privileges, rather than protecting resources. This paper suggests a new management model based upon a holistic view of resources and their controls. By acting upon a resource globally rather than upon individual users, the new approach exposes trends, allows for better resource planning, and allows for easy understanding of the impact of changes on a user's ability to accomplish work. A new tool 'Qualm' is the first component of a new system for dynamic resource management that allows for decisions based not upon fixed limits for resource usage, but upon limits that change with usage patterns and demand.

Introduction

Efficient quota management is difficult. As one frustrated admin put it: "I've been making noise for the last couple of years that I think we should increase (at least double, probably more) the default quotas, but the analysis required to figure out where to increase them has been scaring me." This frustration is a product of the limitations of current approaches to quota management; while viewing and modifying quotas for individuals or a segregated sub-section of the user population is relatively simple, it remains difficult to ascertain current file system usage, determine where change is needed, and assess the impact of a change.

This paper presents a new model of quota management designed to address these shortcomings. The model approaches the problem holistically; rather than focusing on privileges for individual users, it focuses on relative resource share and the global effect of change. The result is a paradigm where only changes that assure the integrity of the underlying resource are valid.

Using this paradigm, a new tool, 'Qualm,' was created. Qualm works on top of existing quota systems to provide a simple means of performing global analysis, as well as a framework for making quota changes in a global context. Qualm employs a holistic approach, using multiple graphical formats to display the state of the entire quota system. These displays allow for quick assessment of the state and efficiency of the quota system at a glance, and provide a means for global manipulation of the underlying system.

The Existing World of Quotas

The current most popular, freely available quota management tools, such as the UNIX *quota* [8] utility and the NT quota system [12], solve the problem of

quota management with usage limits for individual users. Users are given two kinds of limits on the amount of disk space they may use: hard and soft limits. Hard limits are absolute and can never be exceeded. Soft limits may be exceeded by the user, but the user must subsequently lower his disk usage below the limit within a given time frame or suffer a consequence. These tools also provide limited facilities for the creation of user groupings, where a user inherits the quota limit of his group. However, these group mechanisms offer little benefit beyond the ability to administer the quota level of multiple users in one place.

These tools, however, suffer from other severe limitations. Changes to the quota system are singular, meaning they are made without regard to the global state of the quota system and the underlying storage. Moreover, these tools offer no easy way of assessing the current state or efficiency of the quota system, thus rendering global decisions difficult.

Commercial products targeted at the Fortune 1000 genre, such as Precise Software solution's *StorageCentral SRM* technology [15] (which will soon be appearing in a future version of Windows, thanks to a strategic alliance with Microsoft) offer a step up over their freely available counterparts by bringing the kind of flexibility that one would expect from an enterprise solution. Using Precise's software, the system administrator can set up five different kinds of quota limits, better divide users up into service groups, and generate several kinds of reports, from current space usage breakdown by file type, to user or group usage reports. The software even provides a facility for some basic trend analysis: a system administrator can view the usage history for an individual user or a disk.

Nonetheless, this approach still focuses on setting individual user limits. In addition, there is no easy

way to visualize the distribution of the quota system in its entirety, or act upon the quota system in that context; all changes are still delegated at the user-level and those changes are made to user quota limits, regardless of the true state of the underlying disk. Finally, gaining the benefits of Precise's StorageCentral SRM requires a complete and costly switch over to their quota management suite. While another one of Precise's products, *QuotaAdvisor* [14], is much more light-weight, much of the benefits of the complete quota management suite are lost. Much was done during the creation of Qualm to avoid these adoption issues and make the integration of Qualm into the existing quota system relatively simple.

In Search of Inspiration

To overcome the limitations of the existing quota management solutions, a new model of quota management was needed. Recent work by Mark Burgess [3] provided an appealing start. Burgess suggests that a system quota is an inefficient strategy for managing a dynamic resource such as disk space. He then emphasizes the importance of global knowledge when selecting an optimal strategy and concludes, "A quota strategy can never approach the same level of productivity as one which is based on competitive counterforce."

While Burgess' work uses game theory [11] to explore this competition as one between the system administrator and individual users, it ignores an important part of this dynamic interaction: the competition between individual users of the system to meet their own needs above everyone else's. Accordingly, the spirit of Burgess' work was incorporated into the core philosophy of the new quota management model by emphasizing users' relative share of a resource, rather than individual limitations on resource quantities. This fosters an element of competition: since users are allotted a percentage of the resource, an increase in their allotment can only come at the expense of another user.

Burgess' work, unlike some other work in convergent system administration [1, 2, 6, 17], treats policy as a mutable thing that must be changed and tuned for optimal performance. This led to the idea of attempting to model the quota management problem

as a problem in control theory, a branch of mathematics that is often used to model electro-mechanical systems in Electrical Engineering [4]. A possible feedback model for a quota system inspired by control theory is shown in Figure 1. In this model, the system administrator defines an operating curve (OC) that describes how the system will respond to disturbances from a user, $d(t)$, to the system. The control function, as defined by the OC curve, then affects the output of the system, which when combined with the continual input of user activity, causes the system to converge to the new desired state.

While the control-theoretic model offered an interesting new way to be able to graphically control how the quota system enforced quotas and responded to aberrant disk usage, it lacked the "global knowledge for decision making" that Burgess' work emphasized. In the end, both these ideas were merged in the formulation of the newly adopted model of quota management.

Finally, there was the challenge of creating an interface that best represented the global state of a quota system. My work in creating *Peep: The Network Auralizer* [9] demonstrated that when digesting a considerable amount of information, the value of the whole can be greater than the sum of its parts; it is more important to convey the general state and trends of the quota system, rather than the individual values that comprise the system.

In addition, Alva Couch's work on visualization of large execution environments when developing *seeplex* [5] and *xscal*. *xscal* provided inspiration for developing Qualm's scalable graphical environment. The algorithms used in *xscal* for visualizing very large numbers of data points proved very relevant in visualizing quotas; a graphical display of an entire quota system needs to be able to plot thousands of data points on a single display.

A New Model for Quota Management

The new model for the quota system combines elements from the competitive model and the control-theoretic model in a novel way. Rather than the traditional approach of viewing quota as assigning a

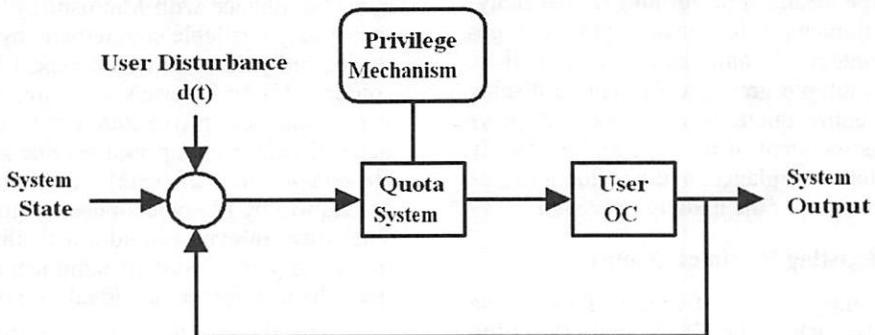


Figure 1: A control theory feedback model for a quota system.

specified amount of disk space to an individual or group of users, all available disk space is treated as a continuous, finite resource, where each user is allotted a fraction of that resource (a simplification which is valid because the smallest unit of measurement, the byte, is very small compared to total the resource size). The system administrator then defines a distribution curve that describes how the administrator thinks the disk resource should be shared, i.e., a certain user population should get more than another user population. This distribution has the constraint that the total area under the distribution curve must be equivalent to the size of the storage resource. Finally, a separate mechanism configured by the system administrator determines user privilege, i.e., where a user gets to sit on the distribution curve.

Figure 2 shows how the model works graphically; this distribution curve allots more space to a sub-section of the population and tapers off for the general population. The X-axis of the graph is then divided up into equal intervals amongst the n -user population, where each user is given a portion of the area under the distribution curve. The user's area then translates into a portion of the storage resource. In this model, the sum of disk space allocated to each user is equivalent to the total storage size, or more formally:

$$\int_0^n d(t) dt = \sum_{i=1}^n \Delta u_i = Total\ Storage$$

where n is the total number of users in the quota system, $d(t)$ is the disk space distribution curve, and Δu_i is the amount of disk space allocated to each user. Figure 3 shows a distribution curve that could be used to implement service levels in existing quota systems under the new model. Here, each step on the distribution curve represents a different quota limit, and its relative length indicates the percentage of the user population have that limit.

Just as in traditional quota systems, a user can use any amount of disk space up to the maximum allotted by their position on the distribution curve. Because of the constraint on the area of the distribution curve, increasing the fraction of disk space for a single user reduces the amount of disk space allotted to every other user. Only changes to a certain user's allotment that do not infringe upon the actual usage of other users (barring explicit action from the system administrator) are considered valid.

The new model offers an important advantage: a user's disk space is decoupled from the actual size of the resource. Thus, if the size of the resource changes, the user's allotted space changes while his relative share remains the same. Taxation of the resource is also reflected by this approach: increasing the number of users (in this case n) automatically decreases the amount allocated to each user. Still, the user's relative privilege remains the same throughout both changes.

This approach also helps separate policy from network implementation; the distribution curve describes

what the system administrator believes proper space allocation should look like, given the needs of his users and the amount of resources available, while the privilege mechanism determines where specific users fit into the system administrator's master plan.

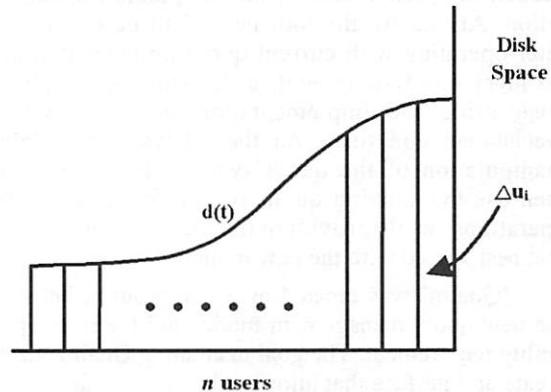


Figure 2: The new model illustrated.

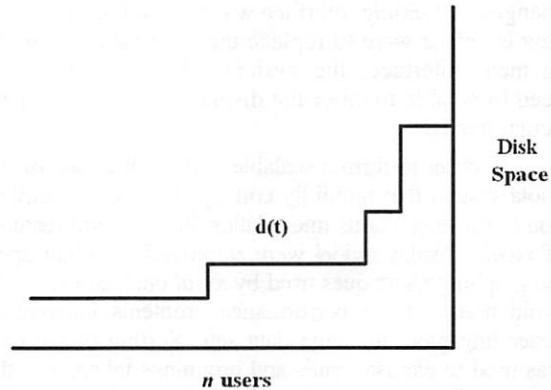


Figure 3: An example of a current quota implementation under the new model.

The model agrees closely with what system administrators currently try to implement as network policy, and incorporates Burgess' idea of competition as a way of maximizing resource efficiency. Here, the distribution curve can be thought of as the system administrator's optimal strategy: a user is given a share of the resource by the system administrator that reflects his given need (as determined by the level of privilege granted by the system administrator) while remaining in accordance with the administrator's general strategy. Giving a single user a larger share of the resource comes at the expense of all other users in the system, which is true of any finite resource, and captures the essence of competition. Consequently, changes are made to a global model, where each change affects the system as a whole, rather than a localized and segregated part.

This has interesting consequences: in the model, the system administrator is no longer trying to limit users but is instead trying to protect and maximize a finite resource. Only changes that maintain the

integrity of the storage and respect other users' use of it are valid.

Towards an Implementation

Some concessions and design constraints were needed to create a successful, adoptable implementation. Above all, the tool needed to be capable of inter-operating with current quota implementations. To meet this requirement, a decision was made to push aside the implementation of the privilege mechanism and focus on the analysis and global manipulation of the quota system. The tool could then use the existing quota system for its back-end operations, while providing the user with an interface that best agreed with the new model.

'Qualm' was created as a compromise between the new quota management model and the inter-operability requirement. The goal in creating Qualm was to create an interface that allowed the system administrator to quickly and easily visualize the state of his quota system, assess problem areas, and make global changes. A flexible interface was also important; if the new interface were to replace the existing quota management interface, the system administrator would need to be able to tailor the displays to reflect his particular needs.

In order to form a scalable view of the state of the quota system that faithfully conveys the global distribution of the data and its interrelationships, several features of *xscal*'s display model were employed. Building upon the graphing techniques used by *xscal* enabled Qualm to avoid many of the performance problems inherent in generating plots for large data sets. Sorting of the data was used to expose trends and groupings inherent in the data set. Qualm moves beyond *xscal*'s abilities in this regard by allowing for multiple data fields to be displayed on a single plot, and allowing for hierarchical sorting. Using hierarchical sorting, secondary fields can be sorted with regard to the result of sorting their parent fields, exposing categories within the data, and trends within those categories.

Qualm's displays use a sum of step functions to depict transitions between values. Because of the large number of data points used in creating the display, a

"continuum effect" occurs and these step functions appear to form a smooth distribution curve. Upon closer inspection (using zooming), the steps become more apparent. The step function provides a nice emphasis on the values and transitions between data points without introducing any graphical artifacts. However, the step function is not always ideal when plotting multiple fields on a single graph. In such cases, Qualm provides alternate display mechanisms, such as error bars that extend away from the primary plot, or scattered data plots.

Furthermore, Qualm's capabilities are modular and extensible. Access to the underlying resource, such as quota data or a flat file, is implemented as a module, so Qualm can easily be extended to work on top of other resources. In addition, Qualm's graphing library provides the system administrator with the tools needed to create new graph types or extend existing graph types, so displays can better be tailored towards the administrator's need.

Performing Analysis with Qualm

Running Qualm on the Tufts University EECS network produced some remarkable results. Figure 5 shows a display of block usage for all EECS users. Values of numbers of blocks used appear on the vertical axis and range over all users of the EECS network on the horizontal axis. The axes are sorted in increasing usage values from left to right. The 1-Dimensional frequency plots at the bottom and left side of the graph indicate the frequency of transitions between values in the main plot, where each hash mark represents a transition. The display yields an interesting result: block usage appears to follow a Pareto [13] distribution. A plot of file usage followed the same distribution. Even more interesting was that this distribution remained constant over a period of six months! This strongly agrees with Michael Mitzenmacher's dynamic model of file sizes [10] and reaffirms that file sizes in large networks are indeed statistically predictable.

Next, one can plot quota hard limits against quota soft limits and block usage using a hierarchical sort (as shown in Figure 6). Once again, values appear on the vertical axis and range over all users on the

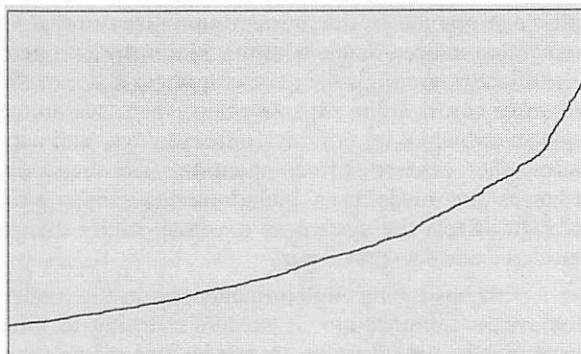


Figure 4a: A Plot at full view.

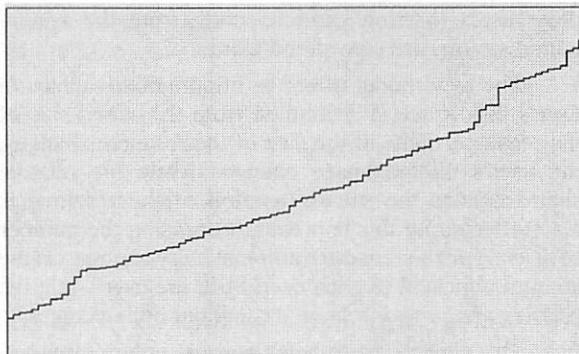


Figure 4b: A zoomed sub-section

horizontal axis. This display allows quick assessment of how quota is distributed across users of the system and how active those users are. The soft quota looked as expected, showing the same distribution as the hard quota settings, except significantly lower. The shape of the quota limits indicate that EECS users tend to fall within four different categories, or four different levels of service. Interestingly, even though Qualm had no knowledge of user groupings, these groupings were implied by the sort order of the graph! Moreover, the block usage yields some very interesting facts: most of our users were given quotas well beyond their needs. At the same time, certain users are over their soft quota limits by a substantial amount and may require additional space.

In order to better assess the effectiveness of the EECS soft quota limits, on-going usage data was accumulated over a period of several months and fed into Qualm. Figure 7 shows a plot of soft limits and block usage with error bars extending from the block usage trace (color coded in reality) indicating the maximum deviation of that usage value over the course of the period. The plot indicates that users who were given larger soft limits tended to deserve it; the space requirements of those users fluctuated more than any

other category. Other users have exceeded their soft limit at some point and might be good candidates for an upgrade to a higher service level. Finally, the large number of over-subscribed users suggests that perhaps with a little tweaking, a much more efficient configuration could be achieved, minimizing the cost of future storage upgrades.

Resource Manipulation with Qualm

A holistic approach to quota manipulation was adopted when creating Qualm: when making adjustments, rather than concentrating on the details, the system administrator tries to make the global picture “look right.” In the context of Qualm, this translates into making adjustments to the distribution curves on the soft limit and hard limit displays.

An adjustment can be anything from lowering or raising a small sub-section of the curve, to radically changing the entire shape of the curve. In order to adjust the limit of the entire population that presently has a particular limit, adjustments are made by left-clicking on the distribution curve at the level of the existing limit, and dragging the plot up or down. The left-click adjustment affects all users with that same initial limit. Alternatively, the administrator can right-click and

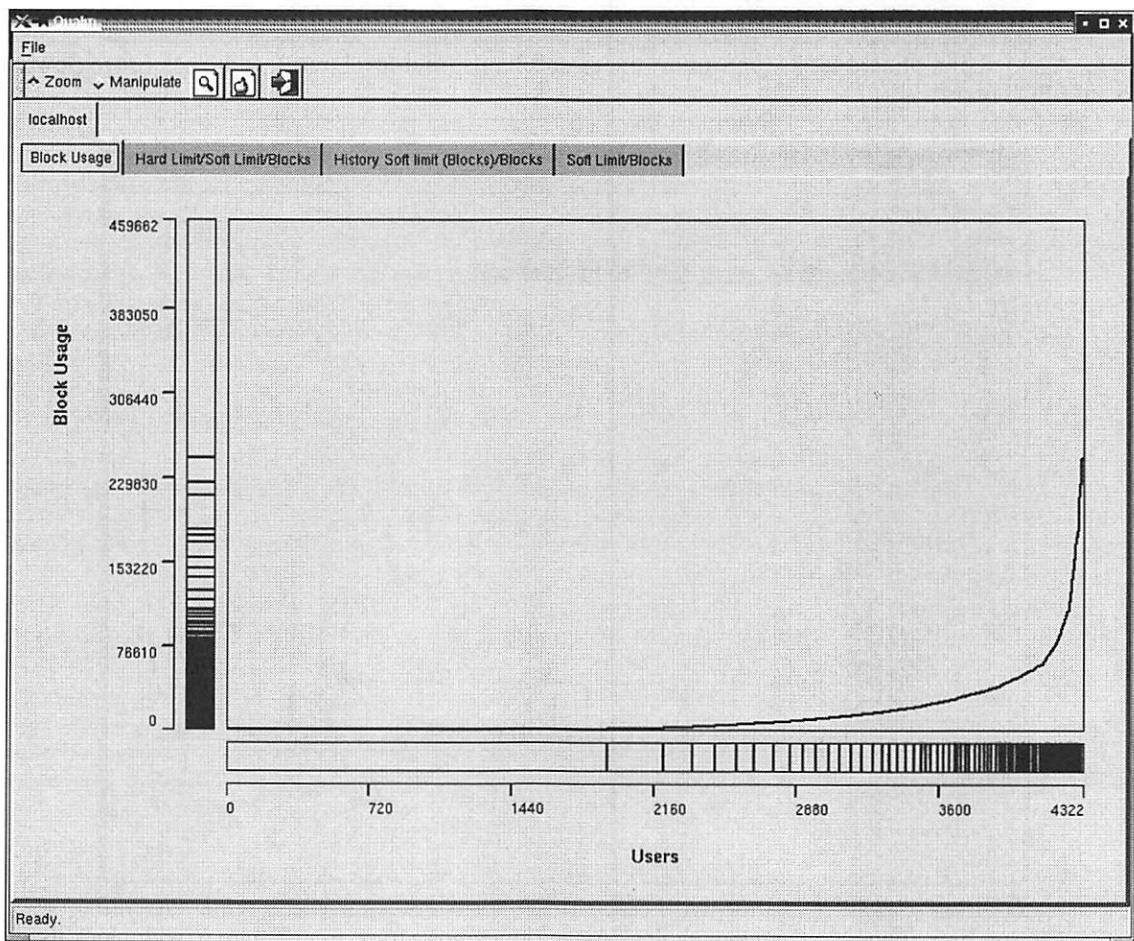


Figure 5: A plot of block usage for all EECS users.

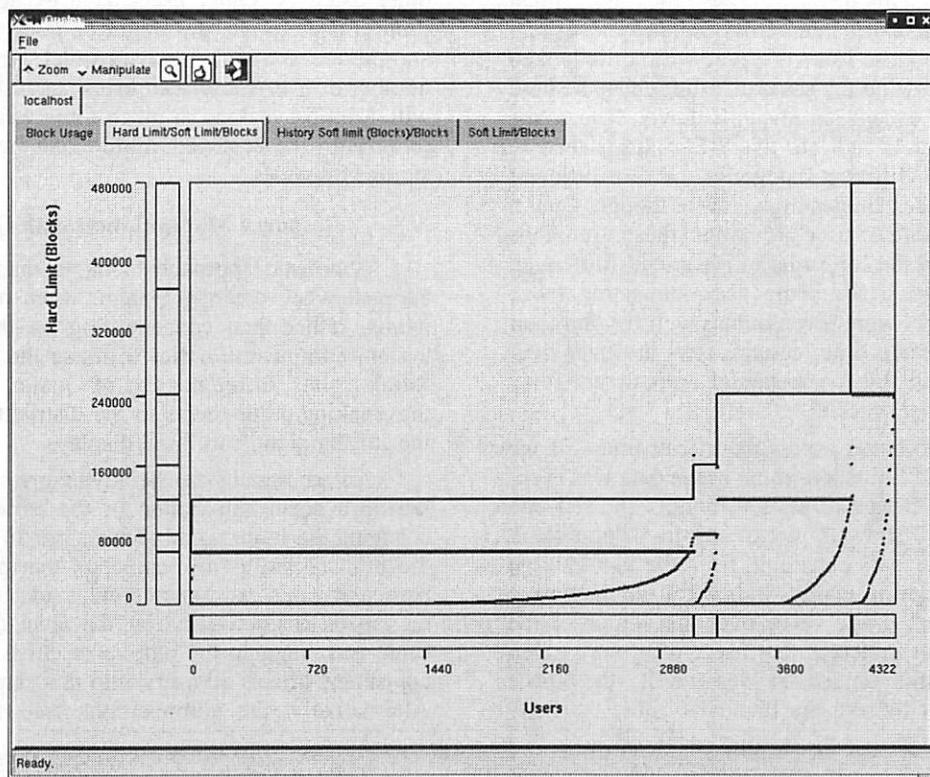


Figure 6: A sorted plot of hard limits (uppermost trace), soft limits (mid-level trace), and block usage (lower most) for all EECS users.

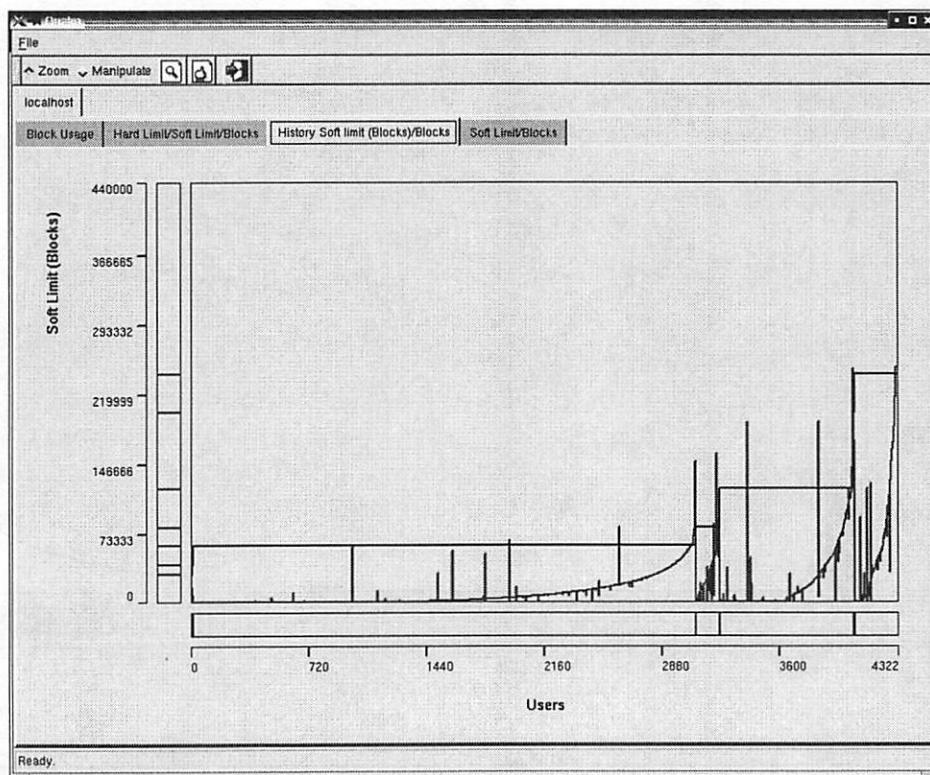


Figure 7: The soft limit (uppermost trace) and block usage (lowermost) trace from Figure 6, with block usage deviations extending from the block usage over time.

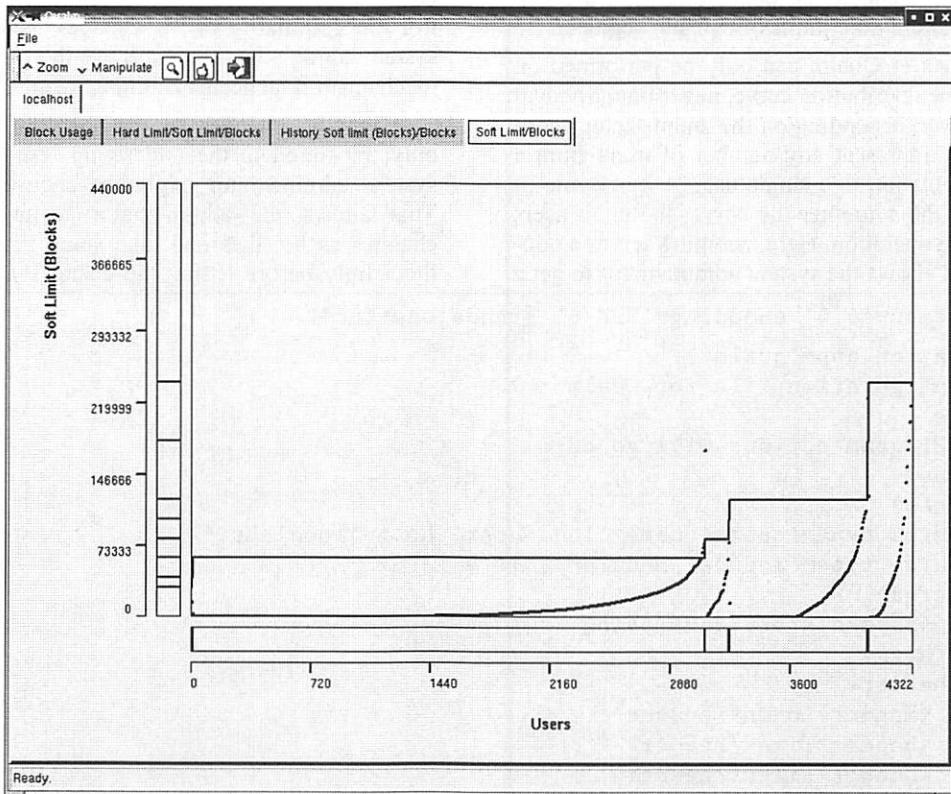


Figure 8a: Manipulation of EECS hard limits to give a higher ceiling to more active users in each usage category.

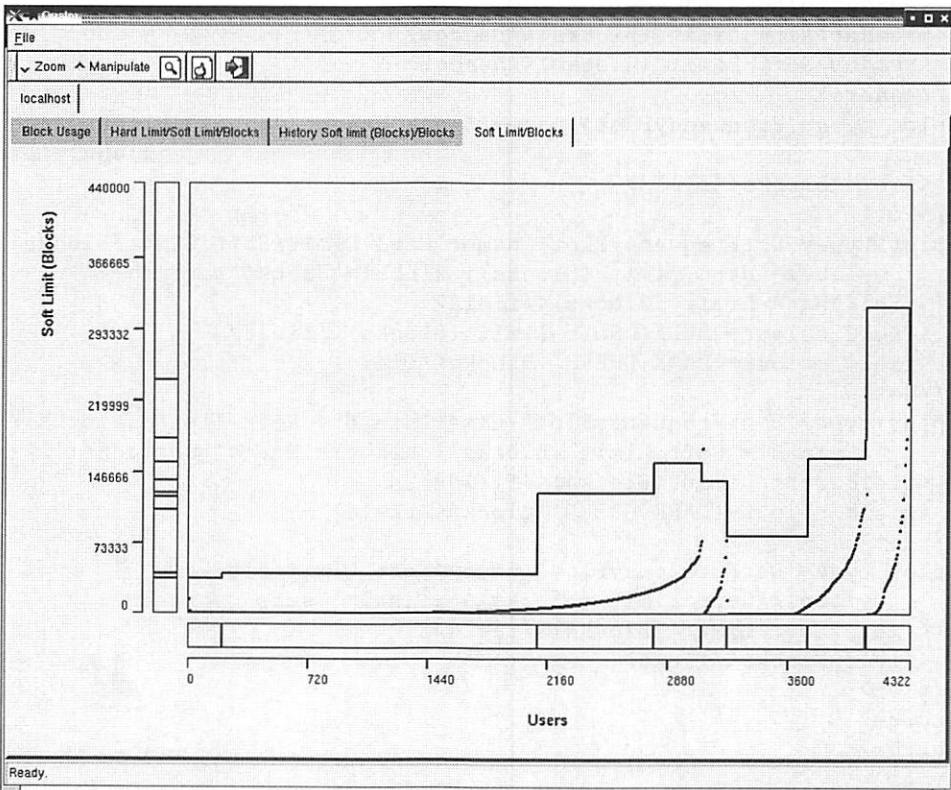


Figure 8b: Manipulation of EECS hard limits to give a higher ceiling to more active users in each usage category, II.

select a sub-segment of the population with a given limit and adjust only the limits of that sub-segment.

Operations in Qualm can only be performed on segments of the distribution curve, never on individual points. However, depending on the zoom-factor, those segments may represent any number of users from a large user population to a single user. A quick rule of thumb is that the smoother the curve, the more users affected by the operation. Here, zooming serves a double purpose: it allows the system administrator to get a

closer look at the underlying distribution and to control the granularity of his changes. Additionally, the system administrator can still use the traditional quota mechanism if he deems zooming insufficient.

Finally, changes to the distribution curve are only transferred to the underlying resource when the system administrator explicitly chooses to commit. This allows the administrator to make as many changes as he likes and take some time to examine them fully before letting the changes go live. It also

```
<?xml version='1.0' encoding='UTF-8' standalone='no'?>
<configuration app='qualm'>
    <!-- Configuration file for Qualm -->
    <options>
        <!-- Program options will go here -->
    </options>
    <resources>
        <resource type='quota' name='localhost' host='localhost'>
            <source type='module' module='File'>
                <args>
                    <path>history.dat</path>
                </args>
                <headers>
                    <header>Login</header>
                    <header>Time</header>
                    <header>Files</header>
                    <header>Hard Limit</header>
                    <header>Soft Limit</header>
                    <header>Blocks</header>
                    <header>Hard Limit (Blocks)</header>
                    <header>Soft Limit (Blocks)</header>
                </headers>
                <plot type='FrequencyPlot' name='Block Usage' ylabel='Block Usage'
                      xlabel='Users'>
                    <field>Blocks</field>
                </plot>
                <plot type='DotFrequencyPlot' name='Hard Limit/Soft Limit/Blocks'
                      ylabel='Hard Limit (Blocks)' xlabel='Users' sorted='1'>
                    <field>Hard Limit (Blocks)</field>
                    <field colour='BLUE'>Soft Limit (Blocks)</field>
                    <field colour='DARK GREEN'>Blocks</field>
                </plot>
                <plot type='HistFrequencyPlot' name='History Soft limit (Blocks)/Blocks'
                      ylabel='Soft Limit (Blocks)' xlabel='Users' sorted='1'>
                    <field>Soft Limit (Blocks)</field>
                    <field colour='DARK GREEN'>Blocks</field>
                </plot>
                <plot type='DotFrequencyPlot' name='Soft Limit/Blocks'
                      ylabel='Soft Limit (Blocks)' xlabel='Users' sorted='1'>
                    <field>Soft Limit (Blocks)</field>
                    <field>Blocks</field>
                </plot>
            </source>
        </resource>
    </resources>
</configuration>
```

Figure 9: An example Qualm configuration file.

avoids a rather expensive operation until absolutely necessary; a single change could easily affect thousands of records.

A Generalized Configuration Format

Qualm uses an XML configuration file to determine how data should be retrieved from a resource, labeled, and displayed. Figure 9 shows an example configuration file that fetches quota system data from a flat file and creates the four displays used for analysis in a previous section.

The **source** tag indicates the data source; Qualm uses resource modules as data proxies. Currently, Qualm supports two types of resource modules: a resource module for interacting directly with the quota system and a resource module for reading data from a flat file. The **args** tag contains parameters which are passed to the module during initialization. The **header** tags indicate how to label the data fields, which may be fields delimited by spaces in a flat file or a list of fields in memory.

The **plot** tags tell Qualm which displays to use and how to generate displays when plotting resource data. The **type** attribute indicates what kind of plot to generate and corresponds to an existing plot object within Qualm's graphing library. The **ylabel** and **xlabel** attributes tell Qualm how to label the axes. The **sorted** attribute indicates whether Qualm should use hierarchical sorting when generating the plot. If the **sorted** attribute is set, Qualm uses the listed order of the fields as the hierarchical sort order. In the "DotFrequencyPlot" example, the hard limit block usage is sorted first, the soft limit block usage is then sorted with respect to the hard limit, and finally the block usage is sorted with respect to the soft limits.

Using this configuration system, a system administrator can add displays or tweak existing displays to meet his needs by simply adding or modifying existing plot tags. Qualm can also load and display data for multiple resources simultaneously by providing multiple **resource** tags. Adding a new **resource** tag adds another page in Qualm's tabular interface, making it easy to look at two different resources from two different locations at once.

Critique

Differentiating between users who may belong in separate groups, but who still have the same quota requirements, is impossible under the current implementation. In the EECS network, we often have students in multiple classes who are given the same disk quotas. Currently, these students fall under the same category, even though it might be convenient to keep them separated for non-quota reasons.

A common example is having two students in different classes, a student in a data structures class and another in a programming languages class, with the same quota allocations. Under the current

implementation, these students become indistinguishable. Both students will lie on the same usage line in a block usage plot, and worse, the relative order of their positions on that line is determined by the data sorting algorithm and thus may fluctuate! The lack of differentiation makes modifying quotas for students in a particular class difficult.

The ideal solution to this problem lies in the implementation of the privilege mechanism described in the new quota management model, and thus a complete implementation of the model. Such a mechanism would need to be able to understand user classes and groupings. This would allow Qualm to better tailor its displays to reflect user grouping (perhaps via color coding or a similar method), keep those groupings together during the sorting process, and allow the administrator to manipulate groupings directly via the display. Such a mechanism would give the administrator the finer grain of control needed to solve this issue. Ultimately, the system administrator can still use the traditional quota system to make changes to specific users.

As with all systems that attempt to increase the efficiency of resource allocation, there is a question of how the system can be defeated. While the analysis with Qualm might suggest a more efficient quota configuration and greatly simplify making those changes, it does punish users who conserve disk space so that they might have that space immediately available when they truly need it. It encourages the mentality to "hoard now, so that I may hoard later," a mentality that is well understood and often employed (sadly, successfully) in the corporate budget world. This problem, however, exists within current quota management systems, and its causes are mostly social factors. Still, because Qualm encourages a more dynamic management model, the influence of these social factors are more significant. The consolation is: while Qualm emboldens change, it also makes it easy to reverse change.

Towards the Future

As Qualm is still in the prototyping stage, the most pressing need in Qualm's current stage of development is user feedback and suggestions. The next phases of Qualm's development will involve working out the kinks and making Qualm a truly usable tool.

Constraining the manipulation of quota levels to keep the area under the distribution curve constant has not yet been implemented at the time of writing. Such a mechanism would play an important role in keeping quota modifications in check; the current interface makes it very easy to lose track of the real scope of a change and make unreasonable changes to quota limits. However, it is crucial for the mechanism to support the concept of over-subscription for inter-operability; existing quota systems greatly over-subscribe storage. Over-subscription may also be desirable for providing users with a bit of breathing room, while keeping a preferable space distribution.

Much future work involves the full implementation of the model; creating an implementation of the privilege mechanism so that users can easily be grouped into classes and different service levels, while keeping the distribution of the resource separate. The privilege mechanism should help solve the most pressing problem with the current Qualm implementation of being able to differentiate classes of users by providing the system administrator with a finer grain of control over how users fit into the quota distribution. Such a mechanism would also make it easier to manage the exceptions that do not fit well into the new quota management model by allowing for a more classic quota management functionality.

Finally, the mechanisms used in Qualm and their implementation are sufficiently flexible that Qualm can be used to analyze any type of network resource with similar properties to disk quota (such as bandwidth, for instance). Future research will explore other areas where the principles used in Qualm may be applied.

Some Lessons Learned

Despite being a fundamental component to modern service networks, disk quota management has changed little since its inception. The traditional focus on controlling user privilege has introduced problems of scalability, making analysis and global manipulation difficult. This paper advocates a new model of quota management that solves these scalability issues; by adopting a graphical approach that emphasizes resource share and distribution, the tasks of determining where change is needed, assessing the impact of that change, and assessing the efficiency of the current quota scheme, are greatly simplified.

Use of Qualm on the Tufts University EECS network has contributed significantly to an understanding of the quota system, and what adjustments are necessary to make it optimally efficient. For the most part, quotas don't affect the usage patterns of the majority of the students; most of the storage allocated to those students will probably never be used and that space might be better allocated to users in higher quota brackets, who tend to place a greater demand on the file system. By graphing patterns of usage over time, it was also easy to determine which users were restricted by their quotas and which users were over-subscribed.

Finally, usage of Qualm has yielded some valuable insight into the nature of our quota systems. It suggests that quota management by category is more effective than artificial user groupings. These categories emerge from similarities found within existing user populations, and provide a better model for controlling global user quotas. Moreover, the strong resemblance of file size distribution to a Pareto distribution suggests that exploiting this knowledge may be the key to building an automated and optimal dynamic quota system in the future.

Availability

Qualm is freely available, open source software. The project is currently under development and I am actively looking for people to get involved with the project, experiment with it, and provide feedback. Qualm is written in pure Python on top of *wxPython* [18] and makes use of the *pyquota* [16] module. To obtain qualm, please visit: <http://qualm.sourceforge.net>.

Acknowledgments

A special thanks goes to Alva Couch for his guidance during Qualm's initial development and pointers to *xscal*'s scalable graphing algorithms. Additional thanks goes to Collin Starkweather for his insightful comments and Etan Lightstone for providing some good food for thought.

Biography

Michael Gilfix was born in Winnipeg, Canada. At age 4, he moved to Montreal, Canada where he spent his younger years. Strangely enough, he currently resides in Austin, Texas. After attending high school at Lower Canada College in Montreal, he went on to receive his B.Sc in Electrical Engineering from Tufts University in Boston, in 2002. He currently designs and develops software for big blue in IBM's Austin software group. He can best be reached by email at mgilfix@eeecs.tufts.edu, or feel free to visit his personal web site at <http://www.eecs.tufts.edu/~mgilfix>.

References

- [1] Anderson, Paul, "Towards a High-Level Machine Configuration System," *Proceedings of the Eighth Systems Administration Conference (LISA)*, SAGE/ USENIX, 1994.
- [2] Burgess, Mark, "A Site Configuration Engine," *Computing Systems*, 8, 1995.
- [3] Burgess, Mark, "Theoretical System Administration," *Proceedings of the Fourteenth Systems Administration Conference (LISA)*, SAGE/ USENIX, 2000.
- [4] *Control Theory and Engineering Links from Theorem.Net*, <http://www.theorem.net/control.html>.
- [5] Couch, Alva, "Categories and Context in Scalable Execution Visualization," *Journal of Parallel and Distributed Computing: Special Issue on Visualization*, Vol 18, 1993.
- [6] Couch, Alva, "SLINK: Simple Effective Filesystem Maintenance Abstractions for Community-Based Administration," *Proceedings of the Tenth Systems Administration Conference (LISA)*, SAGE/ USENIX, 1996.
- [7] Couch, Alva, "Visualizing Huge Tracefiles with Xscal," *Proceedings of the Tenth Systems Administration Conference (LISA)*, SAGE/ USENIX, 1996.

- [8] Elz, Robert, "Disc Quotas in a UNIX Environment," <http://pluto.iis.nsk.su/docs/bsd-4.3/quotas.html>.
- [9] Gilfix, Michael, "Peep (The Network Auralizer): Monitoring Your Network with Sound," *Proceedings of the Fourteenth Systems Administration Conference (LISA)*, SAGE/ USENIX, 2000.
- [10] Mitzenmacher, Michael. "Dynamic Models for File Sizes and Double Pareto Distributions," <http://www.eecs.harvard.edu/michaelm/NEWWORK/postscripts/filesize.pdf>.
- [11] Mobius, Markus. "Introduction to Game Theory," <http://icg.fas.harvard.edu/ec1052/lecture/index.html>.
- [12] *NT Disk Quota System*, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/fileio/fsys_2clf.asp.
- [13] *The Pareto Distribution*, <http://mathworld.wolfram.com/ParetoDistribution.html>.
- [14] *Precise Software Solutions and WQuinn, QuotaAdvisor*, <http://www.wquinn.com/Products/QuotaAdvisor>.
- [15] *Precise Software Solutions and WQuinn, Storage Central SRM*, <http://www.wquinn.com/Products/StorageCentral>.
- [16] *pyquota: A python wrapper for manipulating disk quota*, <http://www.sourceforge.net/projects/pyquota>.
- [17] Traugott, Steve and Joel Huddleston, "Bootstrapping an Infrastructure," *Proceedings of the Twelfth Systems Administration Conference (LISA)*, SAGE/USENIX, 1998.
- [18] *wxPython: A Python GUI toolkit*, <http://www.wxpython.org>.

Application Aware Management of Internet Data Center Software

Alain Mayer – CenterRun, Inc.

ABSTRACT

We have built a comprehensive solution to address the management aspects of deployment and analysis of applications in Internet Data Centers. Our work was motivated by the high total cost of ownership of operating such centers, largely due to the variety of applications and their distinctive management requirements. We have chosen an approach that encapsulates application specific knowledge (*is application aware*) and deployed it in a number of corporate Internet Data Centers. Operations staff found substantial cost reduction in managing applications using our approach.

Introduction

A corporate Internet Data Center consists mostly of Web server farms, application server farms, and database servers. Frequently there is a heterogeneous server environment running Windows 2000, Solaris, Linux, and AIX platforms, often due to corporate mergers and acquisitions. Acquired third-party software can include (1) Web servers such as Apache, iPlanet (SunONE), Microsoft IIS, (2) application servers, such as BEA WebLogic, IBM WebSphere, Microsoft MTS, iPlanet (SUNOne), and (3) database servers such as Oracle and Microsoft SQL servers. In addition, enterprises have a large quantity of in-house developed software (J2EE applications, ASP/JSP pages, COM(+) components, etc.) which need to be deployed and configured on top of the third party software.

There has been excellent prior work on infrastructure deployment and management (see the pioneering paper [TH98] and references therein, such as [R97], or books, such as [LH02], [B00]). These solutions (and some of their tools, e.g., JumpStart, Kick-Start, Ghost, SUP, etc.) mostly focus on more generic, lower level aspects of the infrastructure, such as OS and standard network servers (DNS, mail, etc.). We advocate building on these existing solutions and at the same time creating new management technologies, which are *application aware*. By this we mean that instead of operating only on the level of files and directories, such solutions capture knowledge about an application such as its configuration methods, requirements, and more. Operations personnel can use such knowledge to define methods to deploy, install, upgrade, and start applications. Once defined, these methods can be executed repeatedly and reliably through a “push” method or through a “pull” method (what might be done today using tools such as “rdist” or “cfengine” [CF02], respectively).

It is important to note that the approach of application-aware management technologies extends well beyond the realm of the Web applications in Internet

Data Centers. However, for the sake of concreteness and because of the importance of Web applications, this paper focuses on our efforts to create solutions for Internet Data Centers.

In the next section, we introduce Application Management as an emerging and important field of system administration and then motivate our approach of application-aware management. Subsequently, we present the required building blocks of our approach and shows how they interact with each other to form a comprehensive solution. The next section shows some of the technologies needed to realize the building blocks. After that, we quantify some of the benefits seen by operations people using application-aware technology and finally conclude.

Importance of Web Application Management

In every industry, business is moving online. After migrating business information to databases in the eighties and setting up e-commerce applications in the nineties, organizations of all sizes are relying on web applications for core business operations. With the advent of Web services, this trend will only be reinforced.

Application management today operates on file, directory, and configuration parameter levels. As most operations managers can attest, application changes are frequently hectic ordeals, involving custom-built scripts (sometimes written only moments before they are first used), best guesses about hardware and software configurations, remote locations, late hours, and over-worked staff. Changes can involve complex combinations of commercial software, in-house applications, and custom scripts. Operations staff is responsible for understanding the requirements for all these components, deploying them quickly and flawlessly, and remembering what they have done at a detailed level.

Apart from the actual deployment, application management includes other substantial challenges. It is often important to detect what has changed and by whom in deployed applications (e.g., the administrator on duty at the time of an emergency made a quick fix

to some Web server configuration file/database, which is not consistent with the overall policy). Similarly, operations staff needs to quickly pinpoint deployment and configuration differences among two servers. Severe reliability problems often lead operations staff to undo and rollback application deployments (we note that not every application can be cleanly rolled-back due to its possible side effects, such as changing the OS state).

Cost of Managing Web Applications

Below are the summary points of a representative Internet Data Center environment. The associated cost of managing application deployments and analysis in such an environment is discussed later.

- 120 servers, 26 applications
- Applications are all running on top of Apache Web servers and either Weblogic or WebSphere application servers.
- Collecting application changes and deploying them once a week.
- IBM consulting project to document manual change processes did not result in any improvements in quality or in cost.
- Costly, time-consuming errors, such as running out of disk space during an application deployment or forgetting to add required database tables during an n-tier application update.
- Pain Manifestation: "My team of seven spent 16 hours on one WebSphere deployment"

Motivating Application Awareness

Here we present some typical workflows involving applications both on the UNIX and Windows platform.

J2EE Applications

J2EE application servers, such as Weblogic, WebSphere, and iPlanet (SunOne) implement their own "logical topology" on top of the network of physical server machines. A "server instance" is a software component that makes one or more J2EE applications available on a physical machine. Each physical machine may host one or more server instances. Each product has its own way of creating, grouping, and managing its server instances.

J2EE applications (e.g., Enterprise Java Beans, EJB's) implement the core business logic (second tier in an n-tier architecture) of most Web-enabled applications. These applications are typically packaged in archive formats, such as EAR (Enterprise Archive) or WAR (Web Archive). These archives contain configuration information in XML. Each vendor extends the basic XML configuration, affecting the way the J2EE application is deployed to the application server. Operations personnel often get these archives from the development organization and might have to open the archive to edit configuration values. The deployment of these applications to a server instance always has to be effected via the responsible administrative console server. Details of the deployment commands differ

among vendors and even among versions from the same vendor. For example, the configuration information for each J2EE application running on WebSphere is stored in a centralized database, which is read and written by the administrative console during each deployment, upgrade, or other change.

Closely coupled with J2EE business logic are the Web applications, the first tier in an n-tier architecture. These applications are deployed onto Web servers (Apache, iPlanet, etc.) and contain JSP pages, static content, and more. They are often deployed together with the second tier as a single logical software component, forming cross-machine dependencies. Furthermore, the Web servers need to be configured to connect to the appropriate application server instances.

Windows Applications

On the Windows platform, IIS is the dominant Web server platform and MTS is the dominant platform for the business logic applications. Deployment of applications (ASP pages, ISAPI filters, etc.) onto IIS (versions 5.x) require each configuration to be reflected in the "metabase," a registry like database resident on each Windows 2000 server. Any configuration information about IIS itself also has to be reflected in the metabase. On the Windows .Net server and IIS version 6.x, the metabase is realized as an XML file. Business logic and transactional components are typically packaged as COM or COM+ components. Deployments onto MTS require the registration of these components with the Windows registry database. Any Web or business logic application might require additional manipulation of the registry database. Within the .Net framework, applications are packaged as assemblies, which have yet another deployment philosophy.

Database Dependencies

Most applications require access to database servers running Oracle, SQL, or similar. While deploying and configuring such servers might not be a frequent operation (relative to changes to applications), the deployment of an application typically does require some manipulation of the database (e.g., tables, stored procedures).

Need for Application Awareness

Given the description of Internet Data Center software above and the illustration of Figure 1, it becomes evident that deploying and managing applications requires a lot of application specific knowledge. This is unlike basic infrastructure management of UNIX-based machines, which has been pioneered and described in [TH98] and the corresponding tools for UNIX or Windows (JumpStart, Ghost, isconf [TH98], cfengine [CF02], etc.). There is an unmet need for technology and solutions to automate the deployment and management process beyond host and OS management, and beyond pushing or pulling files and directories of applications.

Application Aware Management

Application aware management requires technology that can (1) capture knowledge about an application, such as its installation, registration, configuration methods and its dependency management and (2) automate processes based on this knowledge. In the following, we present some of the key components of such an approach, building upon known technologies of [TH98], such as central version control, Gold Server, and basic server (host and OS) set-up tools. See also Figure &2, which summarizes the building blocks described below, some of the building blocks introduced in [TH98], and their dependencies. Some of the building blocks in [TH98] have been merged under “Host&OS” management, which can be argued is a pre-condition for application management as a whole. The building blocks above the fat line were introduced in [TH98]; we created the blocks below that line. Lines between building blocks denote that the block below depends on the block above being realized in the system. The system we built includes all building blocks except the “Host&OS” block.

- An **Application Model** is a data-driven representation of an application (including executables and configuration files, content, etc.) and associated methods to deploy, configure, and analyze the application. A model has to be reusable, so that it can be applied to different server environments (e.g., staging vs production), which might require different deployment or configuration choices. Rather than capturing

“hard-coded” configuration settings of the application, the model contains variables for such settings which the user can instantiate during the deployment process.

- The **Model Builder** automates the creation of application models. It captures deployed applications from a Baseline Server (e.g., a machine in the QA environment), checks them into a central version-controlled repository (“Gold Server” approach), and at the same time creates a base model of the application. The model builder associates the type of the captured application (e.g., J2EE application on Web-Sphere, Web application on IIS) with an appropriate base model, including methods for deployment, analysis, and discovery (see subsequent building blocks). The model builder then allows customizing the base model by adding dependencies and configuration customizations (see subsequent building blocks) to the model. An illustration of workflow enabled by this building block is shown later.
- The **Deployment Manager** automates the deployment process of an application end-to-end. For each checked-in and modeled application, this process assures that the application will be correctly installed on the desired server machines. In other words, the deployment management module forms the runtime system for the modeled methods.
- The **Configuration Manager** determines the desired configuration of an application according

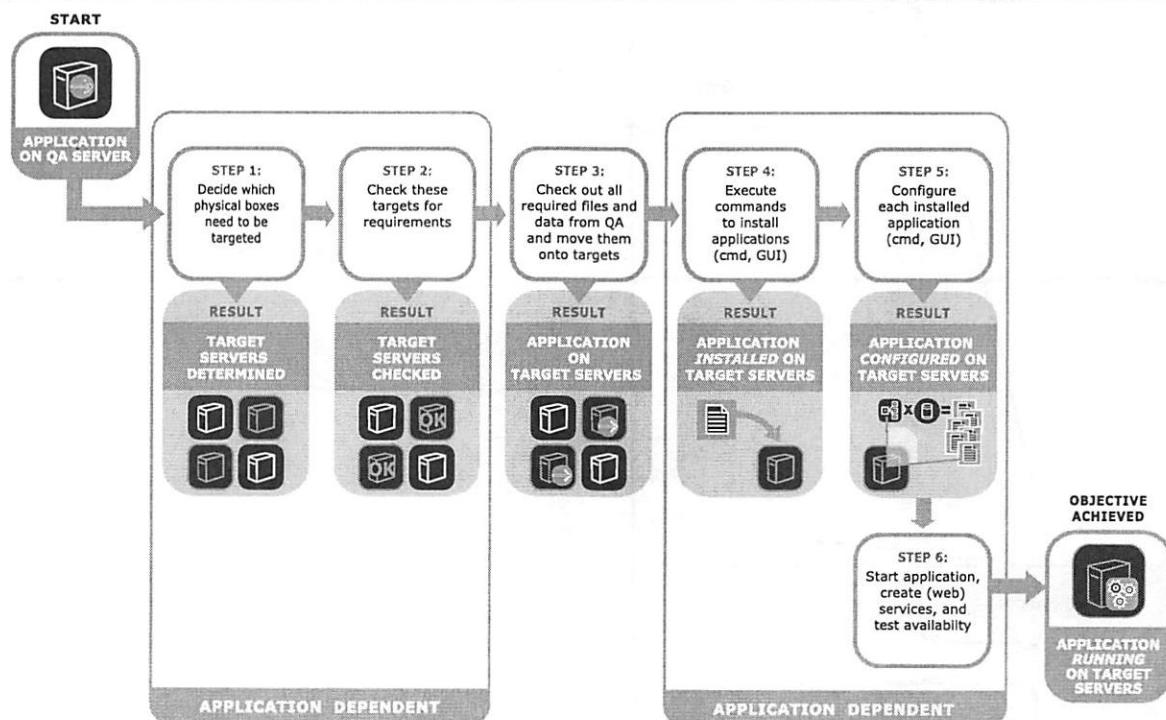


Figure 1: The deployment and configuration of web applications is a complex process, where most steps are application dependent, meaning they differ from application to application (“QA” = quality assurance).

to the environment (e.g., number of CPUs, database connector, and thread-pool of the target server). It then generates the configuration by setting values in appropriate text files, XML files, or modeled methods of the application. In this way, the configuration manager can even write to database-like structures on the target servers, such as Windows registry, IIS metabase, WebSphere data stores, etc.

- The **Dependency Manager** ensures that all modeled requirements for a successful deployment are met (including across different machines). This occurs before the deployment manager makes any changes to the target server.
- **Automated Analysis** pinpoints configuration, version, and other differences between data center servers. This detects the “out-of-band” changes, that are difficult to suppress in most Internet data centers, such as when an operations person changes a configuration value not using any sanctioned tools, but rather by hand, for example during an emergency server recovery procedure.
- **Application Discovery** collects information on what applications are already deployed on which server in the Internet data center. The application model guides the discovery process, as it knows what features indicate the presence

of an application on a server. See [MDEGGH00] for a good introduction to model-driven application discovery.

Figure 3 summarizes how the above components map onto the deployment and analysis process.

Figure 4 illustrates how these technologies fit into a system solution (which we call “CenterRun”). The architecture consists of a master server and remote agents. This solution offers a centralized console to the operations personnel. From this console (command line or Web GUI), applications can be captured from Baseline Servers, application models can be created, and deployments can be executed as captured in the application model. Every action is logged and archived. Installed applications can be analyzed and compared across servers. Below we describe the workflow as offered by the centralized console. We use two concrete and very different applications, a J2EE application on WebLogic (following the sample application “petstore,” one of Sun’s Java blueprint applications) and an n-tier Web application on Windows (following the Microsoft sample application “FMStocks,” see <http://www.fmstocks.com>).

Workflows Enabled by Application Awareness

Workflow 1: Deploy a J2EE Application on WebLogic

1. The Model Builder captures the J2EE application on a Baseline Server:

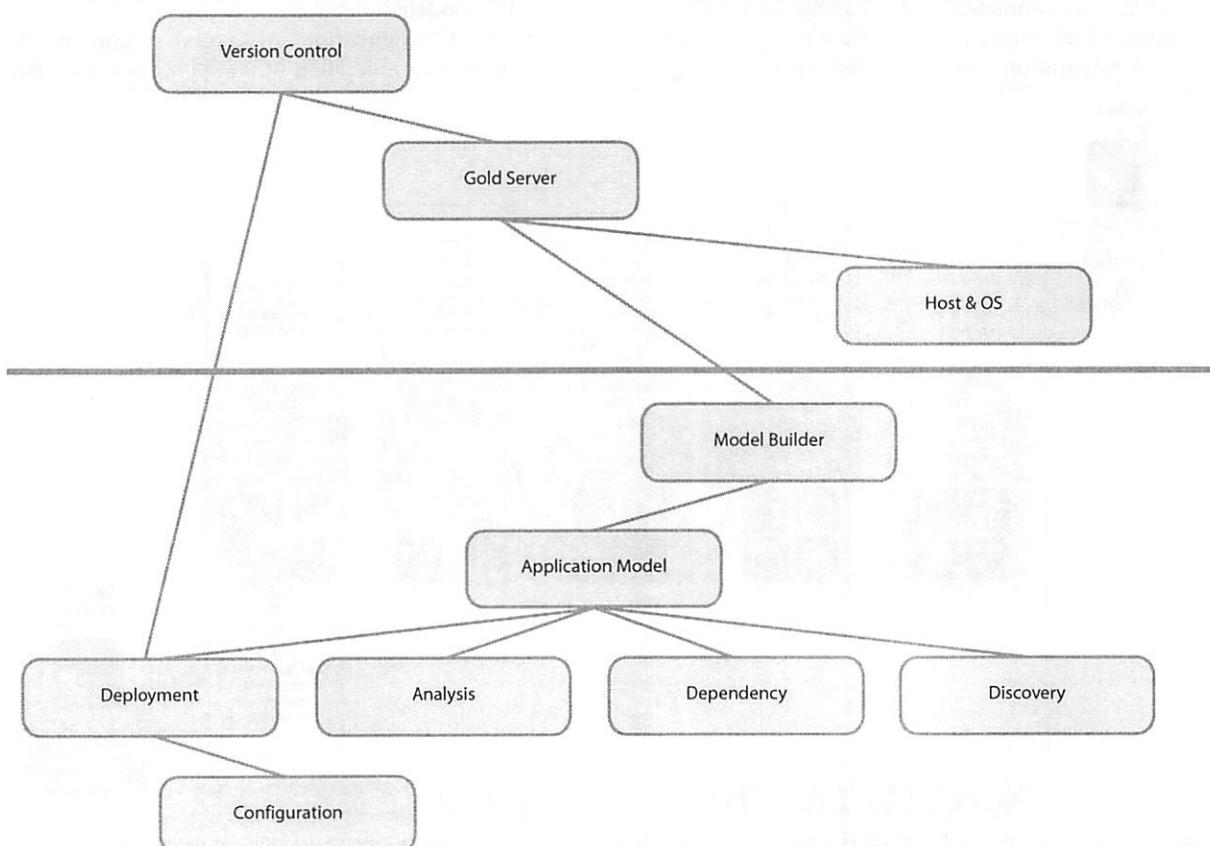


Figure 2: Building blocks for application management.

- a. Recognizes the application as a J2EE application on Weblogic.
 - b. Creates an application model, capturing and describing all the relevant files and archives, such as EAR and WAR.
 - c. Checks these resources into the master server's repository, where they are versioned.
 - d. Captures the relevant configuration information from the Weblogic XML file, "config.xml."
 - e. Uses predefined models to add all the necessary methods to install and uninstall the application to the newly created application model.
2. The Dependency Engine executes checks, such as whether Weblogic 6.0 or higher is actually installed and running, and whether the corresponding Web servers are configured to connect the Weblogic server instances. It does so by querying the remote agent on the target servers.
 3. The Deployment Engine parses the modeled methods for the J2EE application. It then understands which server is the target of the deployment (server hosting the administrative console) and which command-line calls need to be made to the administrative console. It transmits the resources to the agent on the target and has that agent execute the command-line calls.

4. The Configuration Engine determines the configuration values of the J2EE application. For example, the path of the application's home directory on the WebLogic administrative console depends on the WebLogic domain. This install path is modeled as a variable. The Configuration Engine generates the value for this variable by examining the configuration state of the previously installed J2EE server instance.

It is worthwhile noting that Step 1 in the above workflow is typically executed once for each application. As a result, the application and the model is stored and version-controlled on the master server. Steps 2 and 4 are typically executed many times for many different WebLogic target servers. Also, it is very likely that different people within the operations staff execute Step 1 and Steps 2-4. No knowledge about Step 1 is required to kick off the sequence of automated Steps 2-4.

Workflow 2: Compare Two Deployed Instances of the Above J2EE Application

1. The Analysis Engine parses the analysis methods of the model for the J2EE application, which guide it to identify all the relevant configuration settings of the application in the config.xml files of both servers. It then transforms these two files into two new, smaller XML files, containing only this relevant information.

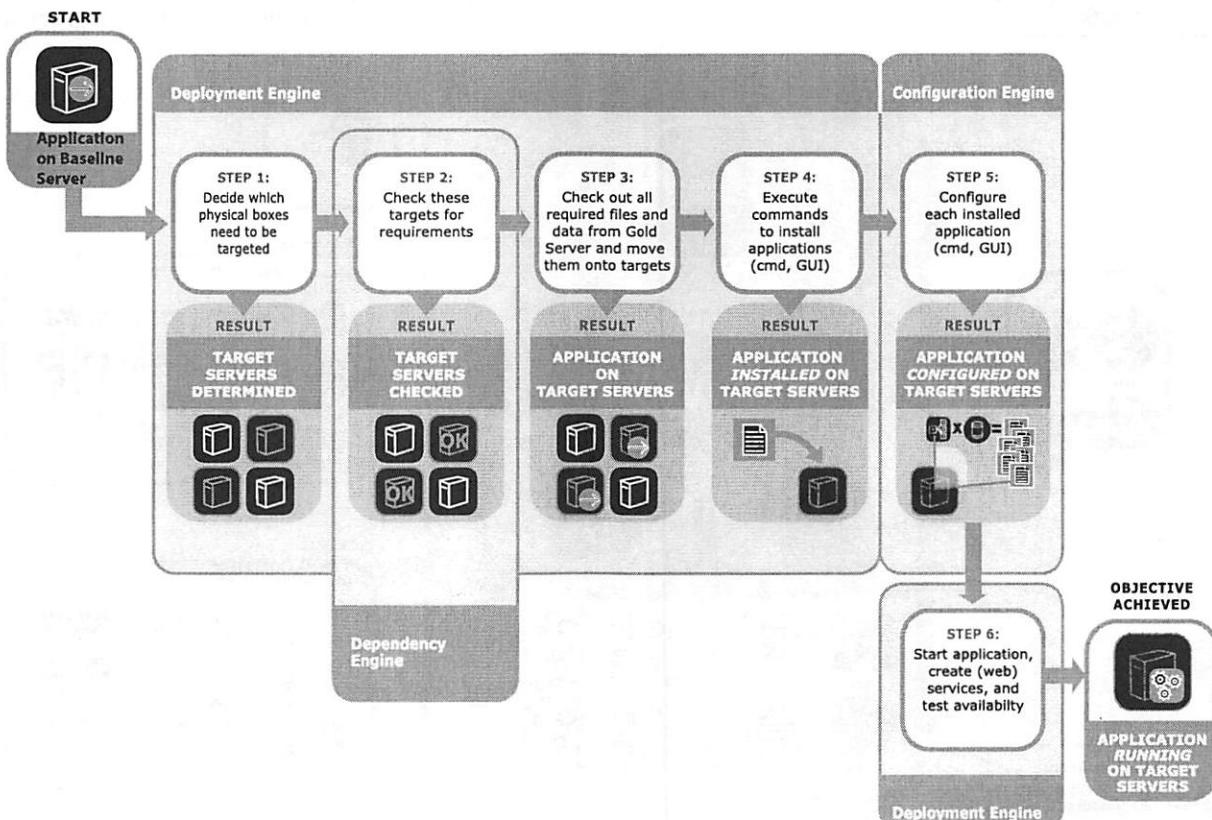


Figure 3: Application aware components can automate the configuration and deployment process from end-to-end, using appropriate application-specific knowledge for each step.

2. The Analysis Engine parses this captured information, compares the two XML files to each other and then presents any differences in a structured way to the user (e.g., the full name of the Weblogic parameter is presented with each differing configuration value).

It is worthwhile noting that the steps in the above workflow use the methods created in Step 1 of Workflow 1. The operations person does not need any knowledge about Step 1 of Workflow 1 in order to execute the above two steps.

Workflow 3: Deploy an N-Tier Web Application on Windows, Consisting of an IIS Virtual Directory, COM+ Components, and a Database

1. Guided by the user, the model builder captures the Web application on a Baseline Server:
 - a. Recognizes the application as a Web application on IIS;
 - b. Creates an application model of the virtual directory, capturing and describing all the relevant content, ASP pages, and ISAPI filters. Figure 5 shows the screen, which lets a user select a virtual directory from the Baseline Server machine (which in Figure 5 is called “*win_ga*”).
 - c. Creates an application model for the COM+ components.
 - d. Creates an application model for the SQL scripts.

- e. Checks the resources of b), c) and d) into the master server’s repository, where they are versioned.
- f. Captures the relevant configuration information from the IIS metabase database and the COM+ catalog and stores them in XML format.
- g. Uses predefined models for each resource type (virtual directory, COM+ component and SQL script) to add to the newly created model all the necessary steps to install and uninstall the application. Figure 6 shows the resources of the completely checked-in FMStocks application. The selection of the virtual directory in Figure 5 resulted in two resources, the virtual directory tree (*FMStocks*) containing content, ASP pages, etc. and the corresponding configuration data (*FMStocks.xml*), which is a resource containing the relevant part of the IIS metabase in XML format. Each resource is listed together with its type. As we will see in the next section, this type determines how the application model is built.

2. The Dependency Engine executes checks, such as whether IIS 5.1 is actually installed and running on the target server or a global ISAPI filter implementing application security is registered

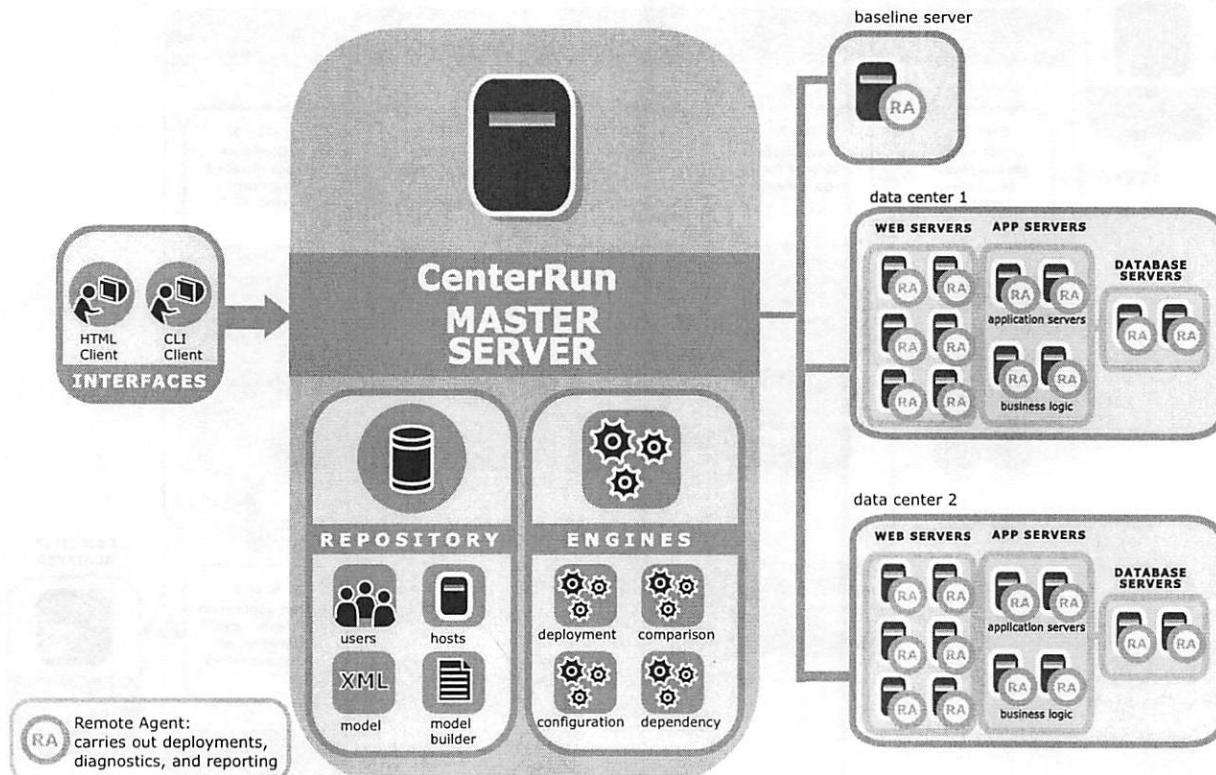


Figure 4: The Master Server (our version of a Gold Server) connects to Remote Agents in different data centers that reside on each managed server.

- in the metabase. It does so by querying the remote agent on the target server.
3. The Deployment Engine parses the modeled methods for the IIS Web application. It then understands how to install the files and ISAPI filters on the target IIS server and which IIS services to shut down at the beginning and restart at the end of the deployment. It also understands how to register the COM+ components and how and on which server to run the SQL scripts. It transmits the resources to the agent on the target server and has that agent execute the installation by making calls into the ADSI API for IIS and COM+ API. The agent also runs the SQL scripts with the appropriate SQL server as target.
 4. The Configuration Engine inserts all the captured configuration data from the XML files into the metabase of IIS on the target server and into the COM+ catalog. It executes the configuration by making calls into the ADSI and COM+ APIs.

Workflow 4: Analyze the Above N-Tier Web Application on Windows for "Out-of-band Changes"

1. The Analysis Engine parses the analysis methods of the model for the IIS virtual directory, which guide it to do the following: (1) identify all the relevant configuration settings of the given application in the metabase of the IIS server and then extract these settings into an XML file; (2) capture all file metadata from the relevant virtual directories and all local ISAPI filter version and configuration data of the given application.
2. The Analysis Engine parses the analysis methods of the model for the COM+ components, which guide it to query the COM+ catalog and capture the resulting configuration settings.
3. The Analysis Engine parses this captured information, compares it to the corresponding information in the master server's repository (where the information relevant at the time of the last deployment is stored), and then presents any

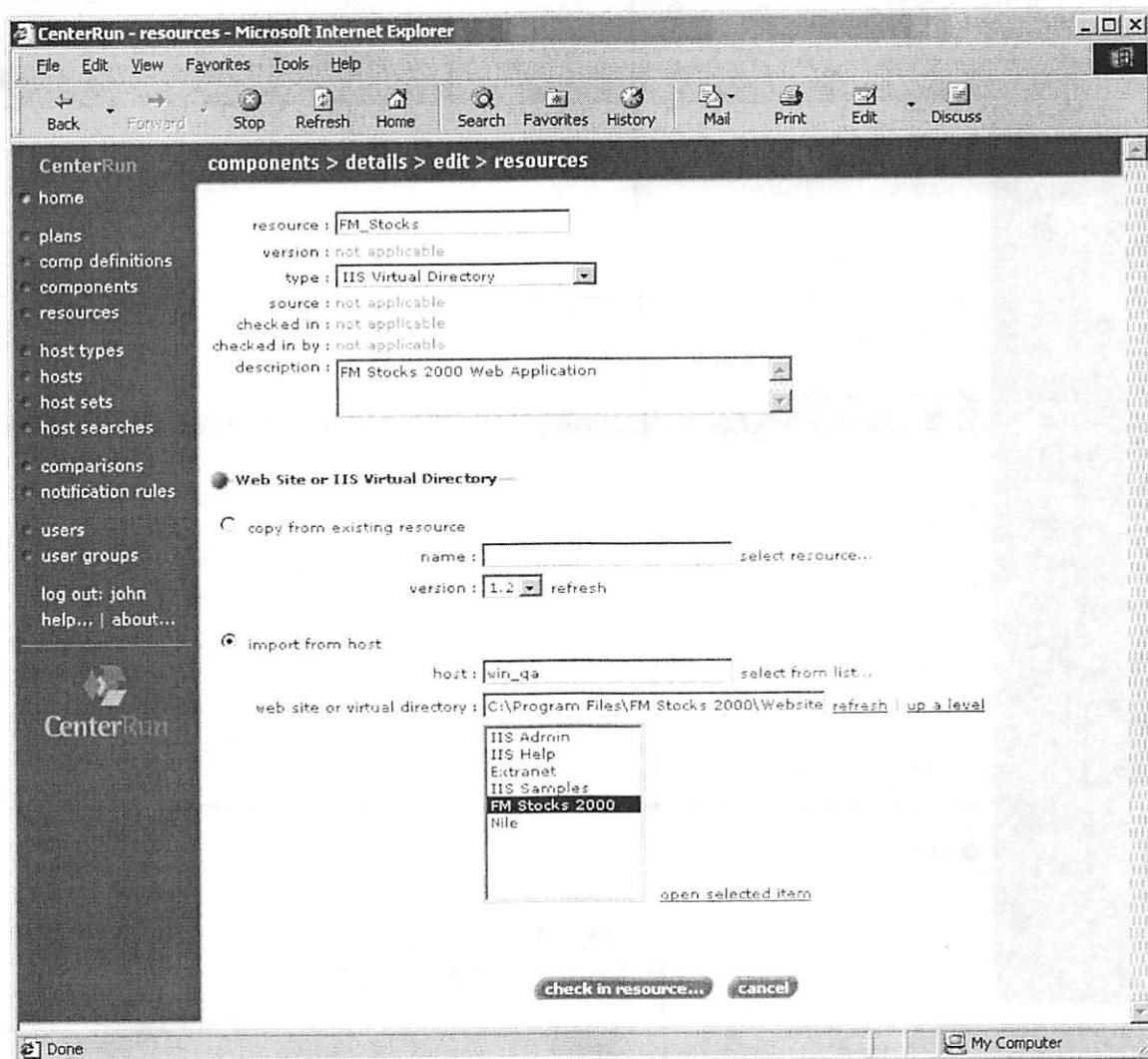


Figure 5: Capture of the IIS virtual directory for FMStocks.

differences in a structured way to the user (e.g., the full name of the metabase field is presented with each difference in the Web application configuration).

Application Aware Technology

In this section, we discuss some of the technology behind the building blocks and functions presented in the last section.

Application Model and Infrastructure

A model needs to capture all aspects of an application: the software features (directories, files, binaries, content, etc.) and the execution steps to deploy, configure, discover, and analyze the application. The model also needs to express the relationships among objects of interest, such as grouping (e.g., software features into an application, target servers into clusters) and

dependencies of one application on other applications being deployed, or dependencies on OS environments on the target server. Consequently, the model is responsible for all of the capturing, storing, and manipulating of application knowledge in the system. At the same time, for a user (system administrator, operations personnel), the model is simply the means to an end, which is the automation of processes. That is why we decided that for common cases, the modeling task should be done by the system. The workflow, introduced in the previous section, allows a user to simply pick resources from a Baseline Server, group them into a modeled and checked-in application and then deploy such application with the “click of a button.” In order to achieve such a workflow, we put the following infrastructure in place:

- A **Component** is a modeled application object, consisting of all the resources and methods

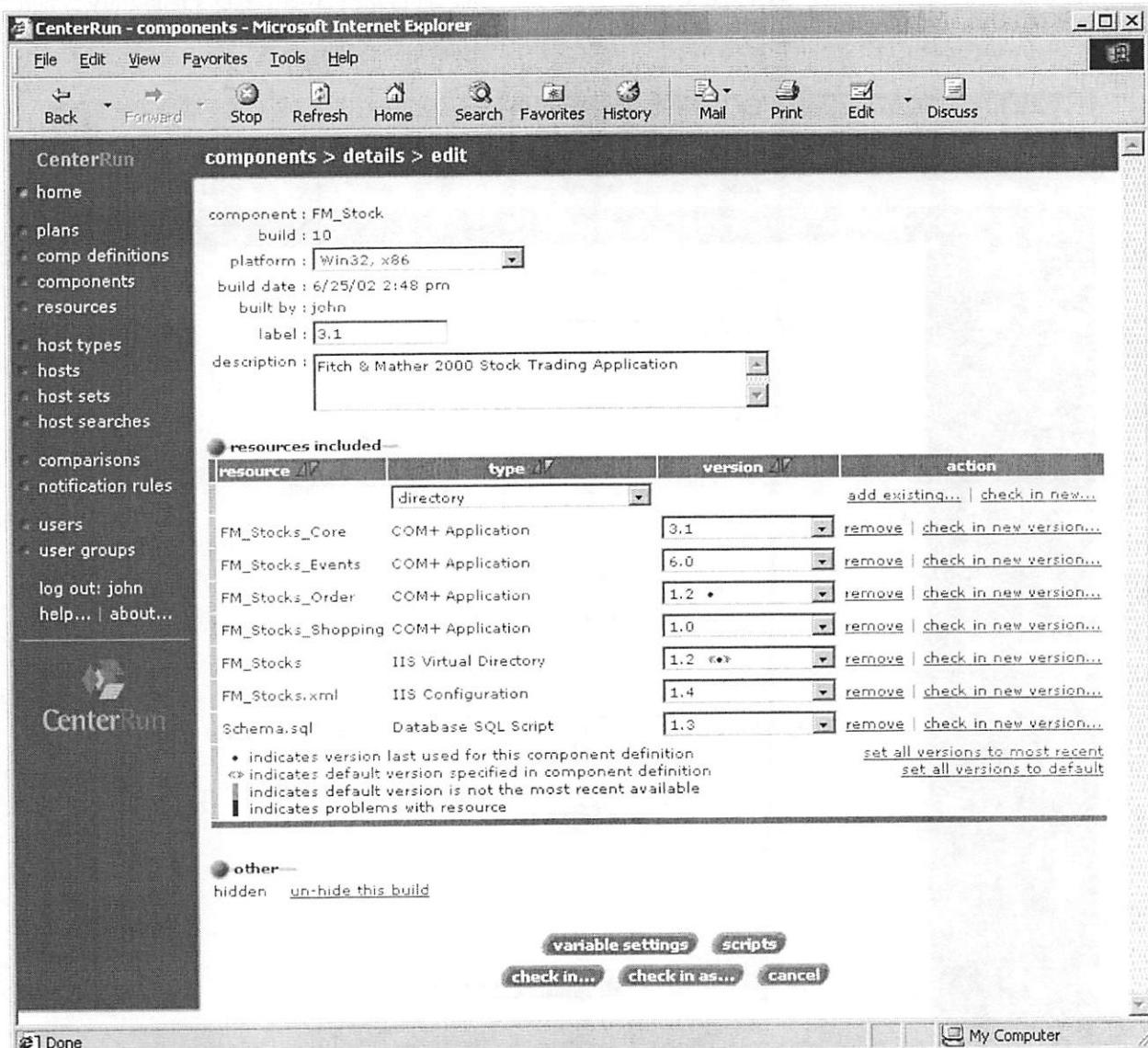


Figure 6: FMStocks application is checked-in and modeled.

(install, uninstall, discover, analyze) of an application.

- A **Resource Type** is a first class object in the system. Each time a user checks a resource into the repository, a *resource type name* (*COM_PLUS*, *WAR_WebLogic*, *IIS_Web_Site*, etc.) is associated with the resource. The resource type associates the corresponding type name with a set of behaviors for deployment, configuration, discovery, and analysis.
- A **Resource Handler** defines the actual behavior. It captures the methods to install, uninstall, discover, and analyze resources of a given type. A *resource handler* is implemented as a *component* itself. These *components* (“system components”) come bundled with the system, as opposed to application components built by the user. A *system component* might have its own resources, which are implementations (scripts, Java classes) of modeled methods for the supported *resource type*. Its methods are accessible by other components being deployed onto the same target server and when calling these methods parameters can be passed. Consequently, the *resource type* really associates a *resource type name* with a *resource handler*. These *system components* are deployed onto target servers transparently to the user, so that they are available to application components at the time of a user initiated deployment or analysis.
- The **Model Language** defines the syntax in which each component is expressed. We chose XML, which is sufficiently readable so that advanced users can manipulate the model directly if needed for customization, rather than just using the model builder.

In the following we illustrate these concepts with some model fragments.

```
<?xml version="1.0" encoding="UTF-8" ?>
<component name="complusHandler" description="Installs com+ objects">
    - <resourceList defaultInstallPath=":[install_path]">
        <resource installName="complus" resourceName="ComPlusScripts" />
    </resourceList>
    - - <controlList>
        - <control name="install" description="Installs a com+ object">
            - <paramList>
                <param name="rsrcDescription" />
            </paramList>
            - <execNative>
                - <exec cmd="cscript">
                    <arg value="/Job:comPlusInstallApp" />
                    <arg value=":[install_path]complus\ComPlus.wsf" />
                    <arg value=":[rsrcDescription]" />
                </exec>
            </execNative>
        </control>
    </controlList>
</component>
```

Listing 1: Fragment of the handler for COM+ resources.

First, we show a fragment of the handler (see Listing 1) for resources of type COM+. This is a *system component*, bundled with the system and thus never has to be manipulated by the user. As depicted below, it consists of a few Windows Scripting Host scripts, which are bundled, in a resource called “ComPlusScripts.” Those scripts are deployed to each target server ahead of time, transparently to the user. The fragment only shows one modeled method, which is the “install” method, responsible for installing any resource of type COM+. This method takes an input parameter (*rsrcDescription*), which determines the COM+ object for the handler. The method then invokes one of the WSH scripts (*Complus.wsf*) via the cscript command shell, passing the input parameter as argument.

Next, we show an *application component* fragment, which corresponds to the FMStocks sample application, mentioned in the previous section. The model builder generates this *component* during the workflow steps 1a-1g, as described in the previous section.

The (complete) component contains a reference to each resource selected by the user in Steps 1b-1d of the workflow. The first two resources (*FMStocks2000* and *FMStocks2000.xml*) are the result of the user selecting the virtual directory. The first resource represents all the content and ASP files, whereas the second resource represents the configuration data of the virtual directory on IIS, as it was set on the Baseline Server. The model builder transparently exports all relevant configuration data from the IIS metabase on the Baseline Server into the *FMStocks2000.xml* file. The third resource represents a COM+ resource. The model builder transparently exports the COM+ object from the Baseline Server into an MSI (Windows Installer) file.

The *installList* XML block (see Listing 2) represents the installation method. The *deployResources* tag

causes for each resource a call to the appropriate resource handler installation method. For the COM+ resource, this causes the installation method in the previous model fragment to be called, with parameters set to appropriate values obtained when the resource was captured. The *installList* XML block contains one step before and one step after the *deployResources* step. These steps stop and re-start IIS on the target server. They are calls to another *system component*, which models Windows services. The model builder generates these calls explicitly, rather than implicitly through *deployResources*. These two steps are not attached to the installation of a single resource, but rather represent global behavior of the deployment of an n-tier Windows application.

In the description above, we introduced our modeling infrastructure. We largely created our own solution. Modeling frameworks in the application management space do exist, among them the following two:

- **CIM:** The common information model [BSTDWW00] is a hierarchical, object-oriented modeling paradigm with relationship capabilities. The core schema of CIM contains objects modeling both basic notions of system management and their relationships. There is an extension schema for applications, containing objects modeling (1) basic notions of software products, features, elements, and actions on these objects (2) their relationships. CIM has been used to model a large part of the Windows platform (the CIM derived model is called WMI) and the Solaris platform (see Solaris WBEM at [DMTF]). CIM is also being investigated for run-time application management in [KKS01].
- **JSR77 ([JSR77]):** A Java Specification Request, which proposes a standard management model for exposing and accessing the management

information, operations, and parameters of the Java 2 Platform, Enterprise Edition components. This proposal covers modeling the basic J2EE notions of J2EEServer, J2EEModule, EJBModule, WebModule, etc., their relationships, and event management. The proposal also discusses mappings of this (specific) model into (the more generic) CIM. JSR77 encapsulates a lot of knowledge about the J2EE world. As pointed out in [FK02], JSR77 lacks expressiveness for describing runtime entities and for representing versions and dependencies.

We found that the above models, while inspiring in many ways were not a good fit for us for an initial implementation. While CIM is a very expressive and extensible modeling framework, it comes with a relatively complex implementation and representation price. It also lacks two key ingredients for us: parameter passing and configuration management. Parameter passing appears to be required for implementing the notion of *resource handler*. Configuration management addresses the issue that the same application will be deployed with different configuration settings depending on the environment and that a model must therefore allow such values to be generated dynamically at deployment time. See the subsequent section for more details. Furthermore, it is important that the model can be conveniently represented to advanced users, who desire to customize the deployment or analysis behavior. CIM does not offer a simple solution for this.

Configuration Management

We decided that configuration management is an area in which we needed to innovate. We designed an extension to our model, which allows the inclusion of variables, and an algorithm (see also Figure 7) which runs at deploy time to instantiate these variables. We also allowed users to replace actual values with

```
<?xml version="1.0" encoding="UTF-8" ?>
<component name="fmstocks" description="FMStocks sample application">
    - -<resourceList defaultInstallPath=":[install_path]">
        <resource installName="FMStocks2000" resourceName="FMStocks2000" />
        <resource installName="FMStocks.xml" resourceName="FMStocks.xml" />
        <resource installName="FMStocks2000Core.msi"
            resourceName="FMStocks2000Core.msi" />
    </resourceList>
    - <installList>
        - <controlService actionPerformed="stop" componentName="servicesHandler">
            - <argList>
                <arg name="serviceName" value="IISADMIN" />
            </argList>
        </controlService>
        <deployResources />
        - <controlService actionPerformed="start" componentName="servicesHandler">
            - <argList>
                <arg name="serviceName" value="W3SVC" />
            </argList>
        </controlService>
    </installList>
</component>
```

Listing 2: The installation method.

variables in configuration files associated with an application. The algorithm determines the configuration data by looking at least at the following *environment characteristics*:

- The characteristics of the server machine (IP address, hostname, network connectivity, processors, etc.).
- The characteristics of the operating system of the server machine (OS type and version, patch level, etc.).
- The characteristics of other software already on the server.
- The characteristics chosen by a user (system administrator) at run-time of the deployment process.

The algorithm reads the model as its first input. Depending on the content of the model, the algorithm then determines the environment characteristics by collecting the relevant data from each target server, from models of other applications provisioned on the target server, and from user generated input. The algorithm uses the collected information to generate values for settings within configuration files of the application and to transform the methods of the model into concrete execution steps, to be executed on each server to install and deploy the application, taking into account all the characteristics collected as described above.

Let us revisit the *petstore* application introduced earlier (see Listing 3). Assume that *petstore* is to be deployed onto WebLogic 6.1. The path of *petstore*'s home directory on the Weblogic administrative

console depends on the Weblogic domain of the J2EE server, onto which *petstore* is deployed. Below is a fragment of an *application component* modeling this fact. The *varList* XML block contains declarations of variables used in this component. The *domain_dir* variable is used to hold the value of the home directory path. The declaration allows defining a default value of a variable. In this case, the default value is a path with a fixed prefix (/opt/bea/wlserver6.1/config) and a suffix, which refers to another variable *domain_name* in another *application component* called *WL61Server*. That component models the WebLogic server instance on the same target server and that variable contains the domain name of the server. The algorithm above instantiates the variable *domain_dir* by examining the configuration state of the installed J2EE WebLogic server.

Deployment Management

The Deployment Manager handles all aspects of the deployment of a modeled application; including efficient WAN distribution of potentially large amounts of data and executing the modeled deployment methods on each target server. This is why we decided to use a local cache (*local distributor*) on LANs, so that a single master and console can control multiple data centers. Another issue is security of the communication. However, none of these issues relate to the main theme of this paper, application awareness.

From an implementation point of view, it is worthwhile to note that JSR 88 (see [JSR88]) is a proposed standard API for the deployment of J2EE

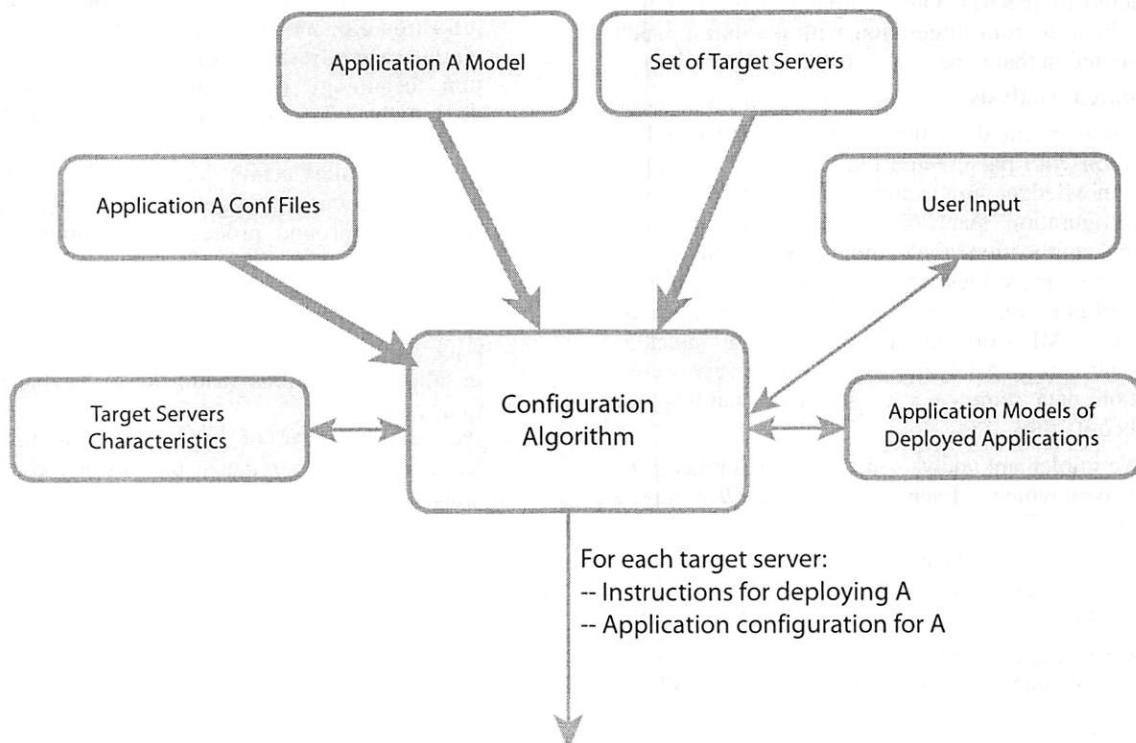


Figure 7: Configuration management algorithm.

applications. The specification aims at defining the contracts that enable solutions from multiple providers to configure and deploy applications on any J2EE platform (e.g., Weblogic, WeSphere, iPlanet, etc.). We need to do deployment across platforms (UNIX, Windows) and well beyond the scope of J2EE applications; still, standardization of this form is a welcome simplification for management tools like ours.

Dependency Management

Dependency information is expressed within our model as relationships among applications. For example, a J2EE application might only be deployable onto WebSphere V4 or newer; WebSphere itself might require the JDK x.y already installed, which in turn requires a certain patch-level of the Solaris OS. On the Windows side, an IIS-based application might require IIS V5 or newer, etc. In CIM, relationships are expressed as objects themselves. We decided to model relationships simply as object attributes, which is a simpler implementation at the cost of some flexibility (e.g., modeling a relationship both directions) and extensibility (e.g., adding a relationship without changing the object in the relationship itself).

An application can consist of several software features. These features might get installed on separate machines (Web server, app server, console for app server). Also yet another set of machines (e.g., database server) might require configuration changes. Our model captures these inter-machine dependencies in the deployment methods and their target servers. Such dependencies are a simplified form of the ones considered in [EK01]. Our approach therefore could greatly benefit from integration with a solution, such as presented in that paper.

Automated Analysis

The remote agent does the analysis of the installed application with parsers and tools, which have application knowledge. For example, in order to analyze the configuration state of an IIS / WebSphere / Weblogic application, the remote agent needs to (1) determine which values are relevant in the corresponding configuration store (metabase / centralized database / XML store) and (2) export these values to the master server. Analyzing Apache Web server configuration data requires a parser understanding the "httpd.conf" file.

We implement analysis in a very analogous fashion to deployment. Each *resource handler* has a

```

<varList>
    <var name="domain_dir"
        default="/opt/bea/wlserver6.1/config/[component:WL61Server:domain_name]" />
</varList>

<resourceList defaultInstallPath=".:[domain_dir]">
    <resource installName="petstore.ear" resourceName="wl61-petstore/petStore61.ear"
              installPath="applications" />
</resourceList>

```

Listing 3: Enhanced *petstore* application.

method, which models the steps to obtain the appropriate configuration data. Analysis consequently implements the runtime environment to that model aspect, which results in the master server obtaining all the relevant data in a well-defined XML format, ready to be analyzed.

Cost Benefits

The organization depicted in the case study of the second section has been spending the following dollar amounts on application management per year:

- \$450K in staff cost for writing deployment scripts, documentation on deployment methods and best practices, etc.
- \$1.1M in staff cost for executing manual changes, deployments, etc. on the servers.
- \$500K in staff cost for emergency response to application failures, deployment failures, etc.

→ This organization spends roughly **\$2,050K on total cost of ownership or \$17K on each of their 120 servers**.

The organization has now adopted our technology in their extranet production environment. Using our technology, they were able to automate the application deployment process from end-to-end. System reliability has improved, both due to shorter maintenance windows and due to less unplanned down time. They were able to free up resources dedicated to application deployment, lowering operating costs. Specifically, they have experienced the following benefits:

1. Drastically reduced time to deploy applications (Apache, WebSphere, and Weblogic base infrastructure and J2EE applications residing on this base infrastructure) through our automation technology (early indications show the quantitative gain to amount to an 80% reduction).
2. Eliminated most errors that typically occurred during deployment (where the remaining issues are mostly around process issues among the operations staff).
3. Significantly reduced firefighting and server rebuilds by using our analysis technology to track "out-of-band" changes (i.e., changes to installed applications made ad-hoc by operations personnel).
4. Increased number of applications supported while eliminating reliance on external consultants.

5. Leveraged reusability of the technology to consistently deploy applications across five different environments.
6. Automatically extracted application builds from Rational ClearCase into our version-controlled repository, reducing errors associated with application builds.

As a result, they could reduce the management costs to the following amounts:

- \$50K in staff cost for authoring and fine-tuning models for their applications.
- \$260K in staff cost for running deployments, upgrades etc.
- \$190K in staff cost for using analysis tools to investigate failures, etc.

→ **Cost of ownership has dropped to roughly \$500K per year, a 75 percent drop from the “before picture,” totaling \$4.2K per server.**

Conclusions

In this paper, we have presented our management approach to deployment and analysis of Web applications. We have argued that an effective solution needs to be application aware and have shown what technology makes up such a solution. Finally, we have described some quantitative implications on the total cost of ownership (TCO) of an Internet Data Center. We believe that application aware management is a new space, with a lot of potential for creating new and exciting technology and solutions.

Acknowledgements

We would like to thank Steve Traugott (Infrastructure Architect par excellence!) for many inspiring discussions. We also acknowledge Avishai Wool (of Lumeta) and Charles Beadnall (of Verisign) for feedback on an earlier draft. Alexander Keller (of IBM) and Steve Traugott were great “shepherds” whose work substantially improved this paper.

Author Information

Alain Mayer has a PhD in Computer Science from Columbia University. After over four years at Bell Labs, Lucent Technologies, he has joined the start-up world. He is currently Chief Technology Officer at CenterRun, Inc, where he guides the research and development of data center management software. He can be reached at alain@centerrun.com .

References

- [B00] Burgess, M., *Principles of Network and System Administration*, Wiley, 2000.
- [BSTWW00] Bumpus, W., et al., *Common Information Model (CIM)*, Wiley, 2000.
- [CF02] cfengine Web site, <http://www.cfengine.org>.
- [DMTF] *The Distributed Management Task Force*, <http://www.dmtf.org>.

[EK01] Ensel, C. and A. Keller, “Managing Application Service Dependencies with XML and the Resource Description Framework,” *Seventh International IFIP/IEEE Symposium on Integrated Management (IM 2001)*, IEEE Press, May, 2001.

[FK02] Frey, G. and R. Kauzleben, “Configuration and Change Management of Java Components Using WBEM and JMX,” JavaOne Conference Talk, 2002.

[JSR77] *The Java 2 Platform, Enterprise Edition Management Specification*, <http://jcp.org/jsr/detail/077.jsp>.

[JSR88] *J2EE Application Deployment Specification*, <http://jcp.org/jsr/detail/088.jsp>.

[KKS01] Keller, A., H. Kreger, and K. Schopmeyer, “Towards a CIM Schema for RunTime Application Management,” *Proceedings of the Twelfth IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM 2001)*, October, 2001.

[LH02] Limoncelli, T. and C. Hogan, *The Practice of System and Network Administration*, Addison-Wesley, 2002.

[MDEGGH00] Machiraju, V., M. Dekhil, K. Wurster, P. Garg, M. Griss, and J. Holland, “Towards Generic Application Auto-discovery,” *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2000.

[R97] Rudorfer, G., “Managing PC Operating Systems with a Revision Control System,” *Eleventh Systems Administration Conference, LISA XI*, 1997.

[TH98] Traugott, S. and J. Huddleston, “Bootstrapping an Infrastructure,” *Twelfth System Administration Conference, LISA XII*, 1998.

Geographically Distributed System for Catastrophic Recovery

Kevin Adams – NSWCDD

ABSTRACT

This paper presents the results of a proof-of-concept implementation of an on-going project to create a cost effective method to provide geographic distribution of critical portions of a data center along with methods to make the transition to these backup services quick and accurate. The project emphasizes data integrity over timeliness and prioritizes services to be offered at the remote site. The paper explores the tradeoff of using some common clustering techniques to distribute a backup system over a significant geographical area by relaxing the timing requirements of the cluster technologies at a cost of fidelity.

The trade-off is that the fail-over node is not suitable for high availability use as some loss of data is expected and fail-over time is measured in minutes not in seconds. Asynchronous mirroring, exploitation of file commonality in file updates, IP Quality of Service and network efficiency mechanisms are enabling technologies used to provide a low bandwidth solution for the communications requirements. Exploitation of file commonality in file updates decreases the overall communications requirement. IP Quality of Service mechanisms are used to guarantee a minimum available bandwidth to ensure successful data updates. Traffic shaping in conjunction with asynchronous mirroring is used to provide an efficient use of network bandwidth.

Traffic shaping allows a maximum bandwidth to be set minimizing the impact on the existing infrastructure and provides a lower requirement for a service level agreement if shared media is used. The resulting disaster recovery site, allows off-line verification of disaster recovery procedures and quick recovery times of critical data center services that is more cost effective than a transactionally aware replication of the data center and more comprehensive than a commercial data replication solution used exclusively for data vaulting. The paper concludes with a discussion of the empirical results of a proof-of-concept implementation.

Introduction

Often data centers are built as a distributed system with a main computing core consisting of multiple enterprise class servers and some form of high performance storage subsystems all connected by a high speed interconnect such as Gigabit Ethernet [1]. The storage subsystems are generally combinations of Network Attached Storage (NAS), direct connected storage or a Storage Area Network (SAN). Alvarado and Pandit [2] provide a high-level overview of NAS and SAN technologies and argue that these technologies are complimentary and converging.

In order to increase the availability of such a system, the aspects of the systems' reliability, availability, and serviceability (RAS) must be addressed. Reliability, availability and serviceability are system level characteristics, which are invariably interdependent. Redundancy is the way resources are made more available. This redundancy permits work to continue whenever one, and in some cases, more components fail. Hardware fail-over and migration of software services are means of making the transition between redundant components more transparent. Computer system vendors generally address hardware and operating system software reliability. For example, Sun

Microsystems has advertised a guaranteed 99.95% availability for its standalone Enterprise 10000 Servers [3]. Serviceability implies that a failure can be identified so that a service action can be taken. The serviceability of a system obviously directly affects that system's availability. A more subtle concern is the impact of increasing system reliability and redundancy through additional components. Each additional software or hardware component adds failure probabilities and thus any project to increase the availability of a system will involve a balance of reliability and serviceability as well. Network Appliance provides a good example of this tradeoff in the design of their highly available (HA) file servers [4].

Disaster protection and catastrophic recovery techniques are not generally considered as part of a vendor HA solution, but the economic reasons which drive HA solutions [5] demand contingency planning in case of a catastrophic event. In short, HA protects the system; disaster recovery (DR) protects the organization.

Recent technical capabilities, particularly in the area of networking have enabled common-off-the-shelf (COTS) clusters to emerge. This paper begins to examine the same technologies and general techniques used in COTS clusters for their feasibility as techniques to

provide geographically distributed systems appropriate for use as remote disaster protection facilities at reasonable cost. In the paper we define geographically distributed in terms of limited communications bandwidth not a distance measurement.

The goal of this work is to outline a design which provides a DR facility that can be made operational quickly for critical functions, provide a means of verifying DR plans and procedures, minimize data loss during the disaster and provide the basis for the reconstruction of the company's computing base. The premise of the paper is to explore the use of HA cluster technologies to distribute a backup system over a significant geographical area by relaxing the timing requirements of the cluster technologies at a cost of fidelity.

The trade-off is that the fail-over node is not suitable for HA usage as some loss of data is expected and fail-over time is measured in minutes not in seconds. Asynchronous mirroring, exploitation of file commonality in file updates, IP Quality of Service (QoS) and network efficiency mechanisms are enabling technologies used to provide a low bandwidth solution for the communications requirements. Exploitation of file commonality in file updates decreases the overall communications requirement. IP QoS mechanisms have been designed to add support for real-time traffic.

The design presented takes the real-time requirements out of the HA cluster but uses the QoS mechanisms to provide a minimum bandwidth to ensure successful updates. Traffic shaping in conjunction with asynchronous mirroring is used to provide an efficient use of network bandwidth. Traffic shaping allows a maximum bandwidth to be set, minimizing the impact

on the existing infrastructure and provides a lower requirement for the service level agreement (SLA).

The next section outlines the approach used for providing a geographically distributed system to support continued operation when a disaster has occurred. Then, DR, HA and IP QoS background is provided. The subsequent section provides details and results of a specific proof-of-concept study. Impacts of the DR elements on the production system are examined along with the issues found during the case study. The paper concludes with a summary and discussion of future work on this project.

The Approach

When developing a DR contingency plan, the restoration of *critical* operations of a data center takes priority. This section proposes a design for a "warm backup" site for such operations using existing communication channels or lower bandwidth commercial communication channels. This design is a compromise between the restoration from backup tapes and a fully synchronous DR facility. The third section offers further discussion of DR options. Tape restoration will be required for non-critical services. The backup site is also not kept synchronous with the primary but is synchronized at a point in the past through asynchronous mirroring. This solution is appropriate for organizations that can survive some loss of availability along with potentially the loss of some data updates. The case study was intended to evaluate the feasibility and gain of the proposed DR solution. Specifically, the case study is intended to evaluate the feasibility of deploying the DR system supporting one terabyte of

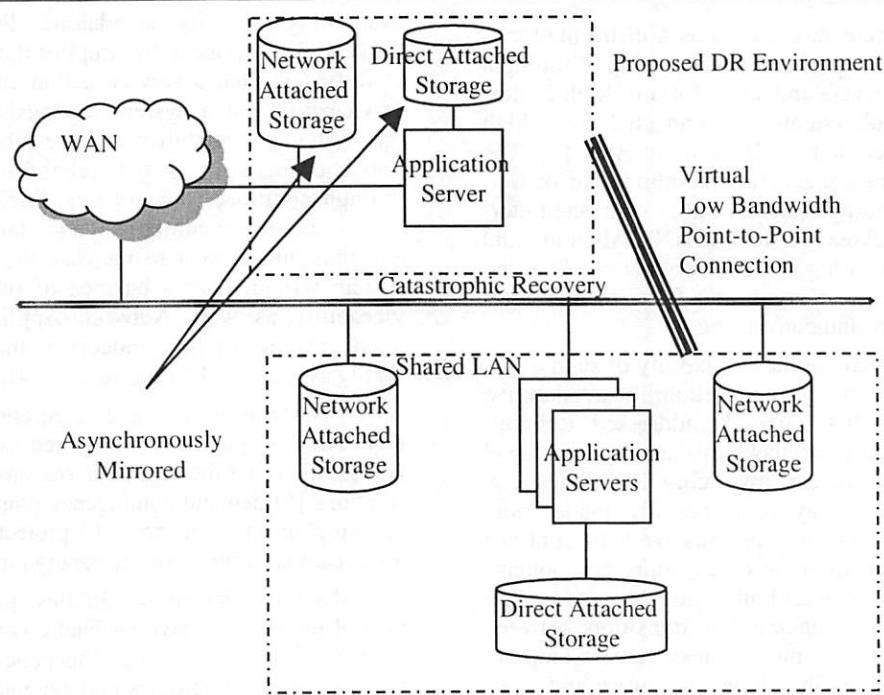


Diagram 1: Proposed DR architecture.

data with no more than 24 hours of lost data updates over an existing T1 (1.544 Mbps) line or another alternative is to use a small portion of a shared WAN.

Topology

The topology of the data center is assumed a centrally located facility of a few to tens of application, file and database servers. Users interface to these services through desktop clients.

The topology of the DR site is assumed to be one of each compatible server interconnected with its primary. Thus, there is a consolidation at the DR site of critical services on many production servers to a single DR server. The data associated with each service must be available to the service's server. All DR servers are interconnected over a LAN. The application consolidation greatly reduces the cost of the DR site. However, the application consolidation complicates the network identity fail-over and is discussed in the Network Identity Fail-over section.

The production environment for the case study is comprised of data service provided via multiple Network Appliance file servers (known as filers) and application services provided by multiple Sun Microsystems Enterprise 5x00 servers and an Enterprise 10000 server. At the DR site, the application services are being replicated on an Enterprise 5000 server and data services are being consolidated to a single Network Appliance F740 filer. Diagram 1 provides an overview of the target architecture. In the proof-of-concept a subset of applications and data are being tested on the local LAN to determine feasibility and SLA requirements for DR deployment. The proof-of-concept test environment is discussed further in the study results section.

Fail-over

A survey of commercial HA solutions (see the Background section) can be generalized as providing for the movement of three critical elements from the failed server(s) to the backup: the network identity, the data, and the set of processes associated with that data. Additionally, a service to monitor the health of each primary service, each backup server, and communications between primary and backup servers is required. This service is known as a heartbeat. In HA solutions this fail-over is normally automated. Since the problem at hand is providing availability in the face of a disaster, which may be predicted and a preventative fail-over initiated or it may be prudent to delay initiation of a fail-over until the culmination of a disaster has occurred, a manually initiated automated fail-over process is used.

Heartbeat

A mechanism to determine system and communications liveliness is required, but the determination is not required continuously as it is for HA. The main issue for this DR site is to keep the data and processes in synchronization to the fidelity of the DR requirements.

Fail-over does not rely on a heartbeat for initiation and synchronization occurs through asynchronous mirroring or shadowing periodically not continuously. Therefore, the determination of system liveliness is required only before the initiation of a synchronization process. The heartbeat mechanism will need to be specific to the file service, mirroring software, and communication technology used. If any errors occur, operational personnel need to be alerted of the problem, but there should be no impact to the production data center.

In the case study, a Korn shell script was written to determine system liveliness. As described in the Data Migration subsection below, the remote mirroring occurs on a volume basis so to determine file system liveliness, prior to initiation of each volume synchronization, the primary and backup filer status is checked and logged via a series of remote status commands (e.g., from Solaris: rsh nacbac sysstat). The status of the primary and backup servers and communications network liveliness is verified and logged by checking the respective network interfaces using various operating system supplied status utilities (e.g., from Solaris: ping, netstat). In the prototype, if any errors occur, e-mail is sent to the personnel involved in the test. In the final production system, alerting operational personnel should be integrated into the system and network management platform.

Process Migration

If the DR site is a mirror of the production data center then commercial shadowing software can be used to synchronize data, applications and system configurations. Since it was assumed that the DR site is not a mirror of the data center, services must be prioritized into separate categories. Each category should have an increasing tolerance for unavailability. These services must be installed, configured and updated along with the primary servers in the data center.

In the case study, only select services are installed on the DR servers and all services must be restarted at fail-over. This fail-over involves bringing the data online with read and write access and a reboot of the servers.

Data Migration

In a DR situation a copy of the data associated with the migrated services must be provided to the DR facility. The integrity of the data must be ensured along with the synchronization of the system as a whole. Commercially, data replication solutions provide a method of supplying and updating a copy of the data at an alternate site. Commercial Data Replication solutions are database, file system, OS or disk subsystem specific; thus, enterprises may be required to use multiple solutions to protect their critical data. The Gartner Research Note T-13-6012 [6] provides a table that differentiates 24 products by the options they support.

In the production environment for the case study, data replication solution must be provided for the network attached storage, direct attached storage controlled

under the Solaris OS and Veritas volume management, along with special considerations for Sybase and Oracle data. The initial proof-of-concept is looking at a subset of the production data environment, specifically the network attached storage with Oracle database data.

Bandwidth Utilization

For this project geographic distribution has been defined in terms of limited communications bandwidth, thus our design seeks to minimize communication requirements. Three bandwidth limiting techniques and compromises are used.

Our first compromise is with the heartbeat. The heartbeat, as previously discussed, is relaxed from a real-time or near real-time monitor to one that only requires activation upon a synchronization event. In the case study, this was once daily and the heartbeat's impact is negligible.

Our second compromise is with data replication. The data is shadowed not synchronously mirrored. This allowed the use of a network efficiency mechanism known as traffic shaping. See Diagram 2.

The objective of traffic shaping is to create a packet flow that conforms to the specified traffic descriptors. Shaping introduces a queuing of network traffic, which is transmitted at a fixed rate resulting in high network utilization. Shaping may change the characteristics of the data flow, by intentionally delaying some packets, thus the need for asynchronous mirroring. Traffic shaped asynchronous mirroring enables data synchronization between the local and remote copies to occur over long periods of time with a constant network impact.

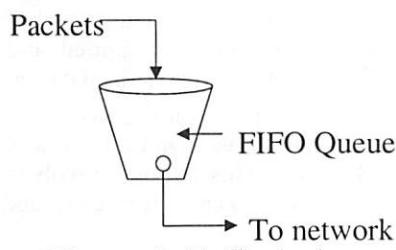


Diagram 2: Traffic shaping.

Even if traffic is shaped at the source, it may become jittered as traffic progresses through the network. To address this jitter, a point-to-point link is required or the traffic should be shaped just prior to entering the low bandwidth link.

Traffic shaping allows a maximum bandwidth to be set minimizing the impact on the existing infrastructure and provides a lower requirement for the service level agreement (SLA). In any communication system using traffic shaping, the finite queue must remain stable.

Queue stability relies on two parameters, the inter-arrival time and the service rate. The service rate of the queue must be greater than data inflow; in our

case, this means setting the maximum data rate allowed on the network high enough. Secondly, the network must be able to successfully transmit the data when serviced out of the queue. IP QoS mechanisms are used to guarantee the necessary bandwidth availability.

Bandwidth availability is greater than required to perform traffic shaped data synchronization, but the high network utilization afforded from traffic shaping will prevent over design to accommodate peak loads over the low-bandwidth link. Traffic shaping and other IP QoS routing mechanisms specifically in a Cisco IOS environment are further discussed in the Background IP Quality of Service section

Our final effort to minimize communications between the local and remote site is an exploitation of file commonality in file updates. Data shadowing products were evaluated which allowed block level updates as opposed to file level updates. It is expected that block level updating will significantly reduce the required communications.

Network Identity Fail-over

The fail-over of the network identity is driven by client availability and for the purpose of DR is more properly stated restoration of client access. If the DR scenario allows for client survivability, the movement of network identity must be addressed. If the DR scenario requires clients to also be replaced, network identity becomes secondary to the client replacement process. An example of client replacement is provided in later in this section.

When a fail-over occurs, the IP address and logical host name used by the Data Center server need to migrate to the DR server. Normally, this is done by reconfiguring the public network interfaces on the takeover server to use the public IP address. This process is complicated by the mapping of the hardware MAC addresses to IP addresses.

The Address Resolution Protocol (ARP) is used to determine the mapping between IP addresses and MAC addresses. It is possible, and common on Sun platforms, to have all network interfaces on a host share the same MAC address. Many system administrators tune the ARP cache used by clients to store the IP-MAC addresses for anywhere from 30 seconds to several hours. When a fail-over occurs, and the IP address associated with the service is moved to a host with a different MAC address, the clients that have cached the IP-MAC address mapping have stale information. There are several ways to address the problem:

- Upon configuration of the DR server's network interfaces, a "gratuitous ARP" is sent out informing other listening network members that a new IP-MAC address mapping has been created. Not all machines or operating systems send gratuitous ARPs, nor do all clients handle them properly.

- The MAC address can be moved from the data center server to the DR server. The clients need to do nothing, as the IP-MAC address mapping is correct. Switches and hubs that track MAC addresses for selective forwarding need to handle the migration; not all equipment does this well. Binding an Ethernet address to an interface is shown using the Solaris naming and configuration syntax:

```
ifconfig qfel ether 8:0:20:1a:2b:33
```

- Wait for the clients ARP cache entries to expire, resulting in the clients realization that the host formerly listening on that MAC address is no longer available and send a new ARP requests for the public IP address.

Movement of the IP address and logical name from the data center server to the DR server is simpler. The use of a virtual hostname and IP address is common. Most network interface cards support multiple IP addresses on each physical network connection, handling IP packets sent to any configured address for the interface. The data center hostname and IP address are bound to virtual hostname and IP address by default. DR and data center server synchronization can occur using the “real” IP address/hostnames. At fail-over, the virtual hostname and IP address are migrated to the DR server. Clients continue to access data services through the virtual hostname or IP address.

Enabling a virtual IP address is as simple as configuring the appropriately named device with the logical IP address, here shown again using the Solaris naming and configuration syntax:

```
ifconfig hme0:1 jupiter up
```

and the “real” addresses are configured one of two ways: on a data center server named europa

```
# ifconfig hme0 plumb
# ifconfig hme0 europa up
```

or on a DR server named io

```
# ifconfig hme0 plumb
# ifconfig hme0 io up
```

The virtual IP is associated with the physical hme0 interface.

If a DR server is a consolidation of several data center servers, virtual IP addresses can be set up for each data center server on the DR server. MAC addresses are assigned per interface so installing an interface for each consolidated server allows movement of the Ethernet addresses. Otherwise, waiting for the ARP cache timeout or a gratuitous ARP can be used.

Client Service Migration

The final task is how to re-establish the clients. Assortments of clients are in use within the case study’s production environment (PCs, UNIX Workstations and thin clients) but for DR, Sun Ray thin clients were chosen. The Sun Ray server software is installed on the DR server to drive the thin clients. A small

number of thin clients are being set-up at the remote site to allow quick recovery with capabilities to add up to 50 units if needed. This is far below the production environment’s normal user load (see Chart 1), but represents a first step towards a return to normalcy.

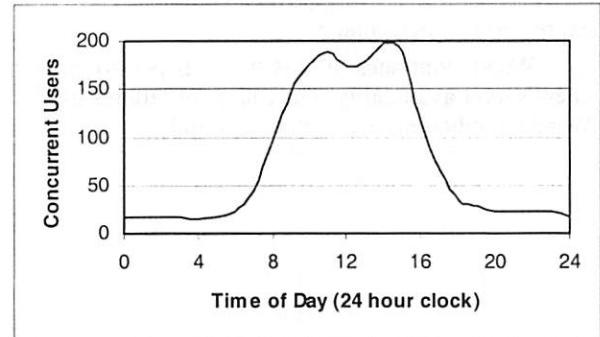


Chart 1: Usage load.

The Sun Ray enterprise system is a Sun Microsystems’ solution based on an architecture that Sun calls the *Sun Ray Hot Desk Architecture* [7]. The Sun Ray enterprise system provides a low cost, desktop appliance that requires no desktop administration, is centrally managed, and provides a user experience equivalent to that of a Sun workstation if servers and networks are properly sized [8]. The Sun Ray appliance is stateless, with all data and processing located on the server. Access is provided to both Solaris and Microsoft Windows 2000 TSE through the Citrix ICA client from a single desktop [9]. The Windows Citrix Servers provide administrative services and are not part of the DR site design but will be required to be rebuilt from tape on new hardware in the event of a disaster.

Background

Failures caused by a catastrophic event are highly unlikely and difficult to quantify. As a result, catastrophic event failures are not normally accounted for in most HA calculations even though their rare occurrence obviously affects availability. The background section begins by defining availability and the levels of availability. DR, HA and the relationship between the two respectively is then introduced. The background section concludes with an introduction to IP QoS mechanisms provided by network routers, with a heavy bias toward QoS features supported in Cisco’s IOS. Cisco routers and switches are used in the case study production environment.

Availability

Availability is the time that a system is capable of providing service to its users. Classically, availability is defined as uptime / (uptime + downtime) and provided as a percentage. High availability systems typically provide up to 99.999 percent availability, or about, five minutes of down time a year. The classic definition does not work well for distributed or client/server

systems. Wood of Tandem Computers [10] presents an alternative definition where user downtime is used to make the availability calculation. Specifically,

$$\frac{\sum_{\text{total users}} \text{user uptime}}{\text{total users}} / (\text{user uptime} + \text{user downtime})$$

expressed as a percentage.

Wood continues in his 1995 paper to predict client/server availability. The causes of failures used in Wood's predictions are summarized in Figure 1.

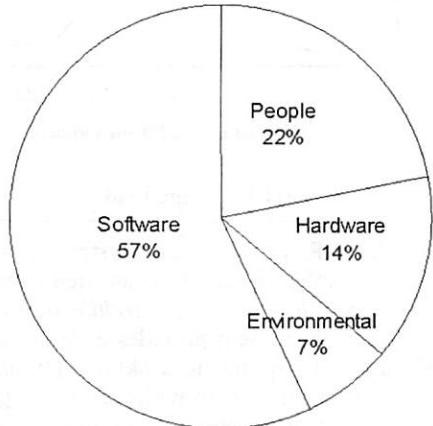


Figure 1: Causes of downtime.

Barr of Faulkner Information Services, relying heavily on research from the Sun Microsystems Corporation, provides a more modern breakdown of the causes of unplanned downtime reflecting improvements in hardware and software reliability [11]. Barr considers three factors as contributing to unplanned system downtime: Process, People and Product. Barr states that Process and People each account for 40 percent of unplanned downtime and Product accounts for the remaining 20 percent. Barr defines unplanned Product downtime to include: hardware, software and environmental failures. The comparison of Wood's and Barr's causes of unplanned downtime demonstrates the trend for vendor software, hardware and environmental reliability improvements to improve overall availability while driving the causes of unplanned downtime more toward their customers' implementations.

Availability Cost

As with any system, there are two ways to improve the cost structure of the system: increase productivity and/or decrease expenditures. As implied by Wood's availability definition, computers and computer systems were created to improve the performance of our work – thus our productivity. The quality of the service provided by the system is an end-to-end statement of how well the system assisted in increasing our productivity.

The way to increase productivity of the user community is to increase the availability of the system in a reliable way. HA solutions provide cost benefits

by pricing out downtime versus the cost of hardware, software and support. The way to decrease expenditures is to increase the productivity of the system support staff by increasing the system's reliability and serviceability.

Availability Levels

Increasing levels of availability protect different areas of the system and ultimately the business. Redundancy and catastrophic recovery are insurance policies that offer some availability gains. A project to increase availability would expect an availability versus investment graph to look similar to the one presented in Figure 2 (adapted from [12]) and can be viewed as having four distinct levels of availability: no HA mechanisms; data redundancy to protect the data; system redundancy and fail-over to protect the system; and disaster recovery to protect the organization. As you move up the availability index, the costs are cumulative as the graph assumes the HA components are integrated in the system in the order presented.

At the basic system level, no availability enhancements have been implemented. The method used of data redundancy will be some form of backup normally to tape. System level failure recovery is accomplished by restoration from backup. The contingency planning for disaster recovery is most often what has been called the "Truck Access Method" (TAM) or a close variant. TAM is briefly discussed in the next section.

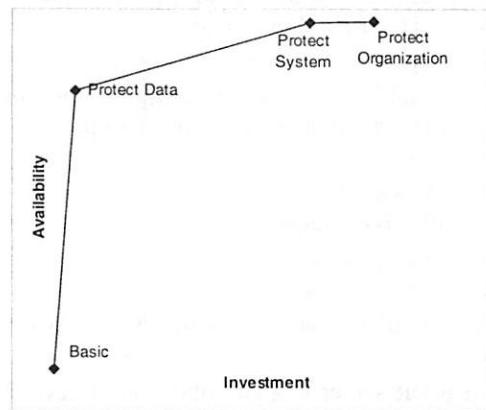


Figure 2: Availability index.

Data protection provides the largest cost benefit of all availability levels. Data protection is important for three reasons. First, data is unique. Second, data defines the business. Finally, the storage media housing data is the most likely component to fail. At the data protection level, a redundant array of inexpensive disks (RAID) [13] solution as appropriate for the environment is implemented. RAID solutions protect the data from loss and provide availability gains during the reconstruction of the data. Volume management software can be added to automate and enhance many

of the system administration functions associated with the RAID disk abstraction.

At the system protection level, redundant HW components are added to enhance the ability to recover after a failure through system reconfiguration and fault tolerance. Automatic system reconfiguration (ASR – logically removing the failed components) and alternate pathing of networks and disk subsystems allow the recovery from hardware failures and coupled with a locally developed or commercial HA solution enables automatic detection and recovery (fail-over) from software service failures.

Disaster recovery is protection for the organization from a business or program-ending event. Disaster recovery differs from continuity of operations in that continuity of operations is proactive and common solutions often rely on transactionally aware duplicate of the production data center to a remote site. Disaster recovery is reactionary. DR assumes some loss of availability and data is acceptable and obtains a cost benefit for this assumption. The differences between the two therefore are cost, reliability and system availability.

Disaster Recovery

DR is a complex issue. DR forces an organization to prepare for events that no one wants to or really can prepare for and to discuss issues that few people are comfortable discussing, such as the loss of key personnel. This paper focuses only on the physical problem of moving critical functions from a data center to a DR site quickly and safely, which is only a portion of disaster contingency planning. There are several sources [14, 15] that discuss surviving and prioritizing [16] a computer disaster from a management perspective.

Kahane, et al. [17] define several solutions for large computer backup centers. The solutions differ by response time, cost and reliability. Among these solutions:

1. Hot Backup – Maintaining an additional site that operates in parallel to the main installation and immediately takes over in case of a failure of the main center.
2. Warm Backup – Maintaining another inactive site ready to become operational within a matter of hours.
3. Cold Backup – Maintaining empty computer premises with the relevant support ready to accept the immediate installation of appropriate hardware.
4. Pooling – A pooling arrangement where a few members join into a mutual agreement concerning a computer center, which is standing-by idle to offer a service to any member suffering from interruption in its computer center. The idle computer center may be employed in the form of “cold” or “warm” backup.

The focus of this project is the creation of a “warm backup” site that integrates an easily testable DR capability into Data Center operations.

Disaster Recovery Approaches

There are generally two common extremes to continuing operations planning, restoration from tape and a fully replicated synchronous backup facility.

The simplest method of DR preparation has been called the “Truck Access Method” (TAM) or a close variant. For TAM, periodic backups of all systems and data are made and stored at a safe and secure off-site facility. The advantage of this method is cost, but there are three main disadvantages affecting reliability and availability. First, the number of tapes can be quite large. A single bad or mislabeled tape can hamper DR extensively. Second, procurement and installation of infrastructure components is time-consuming and anything short of replication of the production data center greatly complicates restoration from tape. Full restoration from tape is also very time consuming. Lastly, testing the DR procedures is complicated and can result in downtime. Extensions to the TAM method include the use of a commercial DR facility or backup pool [17] or the construction of a redundant data center.

At the other extreme, a remote backup facility can be constructed for DR where order-preserving transactions are used to keep the primary and the backup data synchronous [18, 19]. This approach often involves the addition of front-end processors and a mainframe at the remote site. Additionally, the communication link between the data center and remote site will be expensive and potentially can degrade performance of the host applications. In designing a synchronous backup facility, replication is the driving consideration. Wiesmann, et al. [20] provides an overview of replication in distributed systems and databases along with providing a functional model that can be used in designing a specific replication solution.

A common compromise is to employ data vaulting [21] where commercially available data replication solutions mirror or shadow the data to an alternate location, reducing recovery time but mostly reducing risks.

HA Technologies

Replicated hardware solutions have traditionally been used to provide fault tolerance [22]. Fault tolerance refers to design techniques such as error correction, majority-voting, and triple modular redundancy (TMR), which are used to hide module failures from other parts of the system. Pfister [23] and Laprie, et al., [24] can provide the reader more background on fault tolerant systems. Fault Tolerant systems can be classified as primarily hardware or primarily software. Hardware fault tolerant systems tend to be more robust with quicker recovery time from faults but tend to be more expensive. Software systems create very fast recovery by providing a method of migrating a process and its state from the failed node to a fail-over node. Milojicic, et al. [25] provides a survey of the current state of process migration. Fault-tolerant systems are synchronous and the low latency requirements between systems make their use for DR impractical.

HA systems have the distinction from fault tolerant systems in that they recover from faults not correct them. A latent bug in application code which causes a fault is unlikely to re-occur after a fail-over in a HA solution as the application will be re-initialized. This distinction makes HA solutions the preferred solution for software failures and most operational failures. Fault tolerant systems are generally used in mission critical computing where faults must be masked and are often components in HA systems.

The basic model for building a HA system is known as the primary-backup [26] model. In this model, for each HA service one of the servers is designated as the primary and a set of the others are designated as backups. Clients make service requests by sending messages only to the primary service provider. If the primary fails, then a fail-over occurs and one of the backups take over offering the service. The virtues of this approach are its simplicity and its efficient use of computing resources. Servers providing backup services may be "hot-standbys" or themselves providers of one or more primary services. The primary-backup model of HA system provides a good model for the creation of a "warm backup" site.

In building an HA system, ensuring data integrity in persistent storage during the fail-over process is the most important criteria, even more important than availability itself. In general, there are two methods for providing a shared storage device in a HA cluster: direct attached storage or some form of network storage.

Direct attached storage is the most straightforward persistent storage method using dual-ported disk. The issue is one of scalability. As an HA cluster requires disk connection to all primary and fail-over nodes, clusters of greater than four to eight nodes cannot support direct attached persistent storage and require disk systems be deployed in some form of a storage network.

Storage Area Network (SAN) environments are dedicated networks that connect servers with storage devices such as RAID arrays, tape libraries and Fiber Channel host bus adapters, hubs and switches. Fiber Channel SANs are the most common SANs in use today [27]. Fiber Channel SANs offer gigabit performance, data mirroring and the flexibility to support up to 126 devices on a single Fiber Channel-Arbitrated Loop (FC-AL) [28]. SANs are a maturing technology as such there are numerous developing standards and alliances for Fiber Channel SAN design. Furthermore, distance limits for a SAN are the same as the underlying technologies upon which it is built. An emerging standard, Fiber-Channel over TCP/IP (FCIP) [29] has been proposed to the Internet Engineering Task Force (IETF). FCIP offers the potential to remove the distance limitations on SANs.

An alternative to SAN environments is NAS (Network Attached Storage) networks. NAS networks

also connect servers with storage devices but they do so over TCP/IP via existing standard protocols such as CIFS (Common Internet File System) [30] or NFS (Network File System) [31].

The use of a storage network to provide data access during fail-over is sufficient for HA but does not provide for the separation of resources necessary in DR. DR needs a copy of the data at a remote location. The addition of a data mirroring capability is a potential solution. The mirroring process of the persistent storage can be synchronous, asynchronous or logged and resynchronized. Local mirroring, also known as RAID 1, consists of two disks that synchronously duplicate each other's data and are treated as one drive. One solution to providing a remote copy of the data would be to stretch the channel over which the data is mirrored. A single FC-AL disk subsystem can be up to 10 kilometers [28] from the host system. For some disaster contingency planning, ten kilometers may be sufficient. Channel extenders offer potential distances greater than ten kilometers [21].

An alternative to synchronous mirroring is asynchronous mirroring (also known as shadowing). In asynchronous mirroring, updates to the primary disk and mirror are not atomic, thus the primary and mirror disk are in different states at any given point in time. The advantage of asynchronous mirroring is a reduction in the required bandwidth as real-time updates are not required and a failure in the mirror does not affect the primary disk. The two basic approaches to asynchronous mirroring are: to take a "snapshot" of the primary data and use a copy-on-write [32] mechanism for updating both the mirror and the primary data; or to log updates [33] to the primary data over a period of time then transfer and apply the log to the mirror. The result of asynchronous mirroring is that the data and the mirror are synchronized at a point in the past.

Commercial HA Solutions

There have been small investigations [35] into the formal aspects of the primary-backup system but the traditional two to a few node HA systems have been widely used commercially. Many commercial cluster platforms support fail-over, migration, and automated restart of failed components, notably Compaq, HP, IBM, and Tandem [23], Sun's Full Moon [36] and Microsoft's Cluster Service [37]. All of the commercial cluster platforms mentioned offer only single-vendor proprietary solutions. Veritas Cluster [38] offers a multi-vendor HA solution.

Commercial products like BigIP [39] from F5Networks or TurboLinux's TurboCluster [40] have been introduced where clustering is used for load balancing across a cluster of nodes to provide system scalability with the added benefit of high availability. These systems are employing novel approaches to load balancing across various network components and IP protocols. The use of clustering across geographically distributed areas is gaining support for

building highly available Web Servers [42, 43, 44]. The geographic distribution of the Web Servers enhances high availability of the Web interface and provide for virtually transparent disaster recovery of the Web Server. The DR issue with the design is with the back-end processor. In the Iyengar, et al. [43] article, the back-end processor was an IBM large-scale server in Nagano, Japan without which the Web Servers provide only static potentially out of date data.

High Availability and Disaster Recovery

On the surface, DR would seem to be an extension of HA. HA's goal is not only to minimize failures but also to minimize the time for recovery from them. In order to asymptotically approach 100% availability, fail-over services are created. One DR strategy would be to create a fail-over node at an appropriate off-site location by extending the communications between the clustered systems over geographic distances

However, DR and HA address very different problems. Marcus and Stern [12] made four distinctions. First, HA servers are colocated due to disk cable length restrictions and network latency; DR servers are far apart. Second, HA disk and subnets are shared; DR requires servers with separate resources. Third, HA clients see a fail-over as a reboot; DR clients may be affected also. Finally, HA provides for simple if not automatic recovery; DR will involve a complex return to normalcy.

Furthermore, commercial HA solutions assume adequate bandwidth, often requiring dedicated redundant 10 or 100 Megabit channels for a "heartbeat." Data center performance requirements often require Gigabit channels for disk access. Even if network latency and disk cable length restrictions can be overcome with channel extension technologies [21] and bandwidth, the recurring communications cost associated with providing the required bandwidth to support HA clusters over geographic distances is currently prohibitive for most organizations.

The level to which HA technologies can be cost effectively leveraged in a DR solution offers some simplification and risk reduction of the DR process. In order to cost effectively use HA technologies in DR, the high bandwidth communication channels must be replaced with low bandwidth usage. Our focus is to minimize required communications between the primary and the backup, efficiently utilize the available bandwidth and rely on IP QoS mechanisms to insure a stable operational communications bandwidth. The next subsection provides an overview of IP QoS mechanisms.

IP Quality of Service

In order to provide end-to-end QoS, QoS features must be configured throughout the network. Specifically, QoS must be configured within a single network element, which include queuing, scheduling, and traffic shaping. QoS signaling techniques must be configured for coordinating QoS from end-to-end between network elements. Finally, QoS policies must be developed and

configured to support policing and the management functions necessary for the control and administration of the end-to-end traffic across the network.

Not all QoS techniques are appropriate for all network routers as edge and core routers perform very different functions. Furthermore, the QoS tasks specific routers are performing may also differ. In general, edge routers perform packet classification and admission control while core routers perform congestion management and congestion avoidance. The following QoS overview of router support is biased toward what is available as part of Cisco's IOS as Cisco routers are used in the case study environment. Bhatti and Crowcroft provide a more general overview of IP QoS [45].

Three levels of end-to-end QoS are generally defined by router vendors, Best Effort, Differentiated and Guaranteed Service [46]. Best Effort Service, (a.k.a. lack of QoS) is the default service and is the current standard for the Internet. Differentiated Service (a.k.a. Soft QoS) provides definitions that are appropriate for aggregated flows at any level of aggregation. Examples of technologies that can provide differentiated service (DiffServ) in an IP environment are Weighted Fair Queuing (WFQ) with IP Precedence signaling or, under IOS, Priority Queuing (PQ) when only a single link is required [47]. Guaranteed Service (a.k.a. hard QoS) is the final QoS level as defined. Guaranteed Service provides a mechanism for an absolute reservation of network resources. Integrated Services (IntServ) guaranteed service could be configured using hard QoS mechanisms, for example, WFQ combined with Resource Reservation Protocol (RSVP) [48] signaling or Custom Queuing (CQ) on a single link [49] in a Cisco IOS environment.

In a router environment, end-to-end QoS levels are implemented using features provided as part of the router's operating system. These features typically fall into five basic categories: packet classification and marking, congestion management, congestion avoidance, traffic conditioning and signaling. In a Cisco router environment, the Internetworking Operating System (IOS) provides these QoS building blocks via what Cisco refers to as the "QoS Toolkit" [50].

QoS policies are implemented on an interface in a specific sequence [51]. First, the packet is classified. This is often referred to as coloring the packet. The packet is then queued and scheduled while being subject to congestion management techniques. Finally, the packet is transmitted. Packet classification is discussed next, followed by a discussion of queuing and scheduling. Congestion avoidance techniques are used to monitor the network traffic loads in an effort to identify the initial states of congestion and proactively avoid it. Congestion avoidance techniques are not used in this project and will not be discussed further. The IP Quality of Service section proceeds with a discussion of traffic shaping and policing; concluding with a discussion of RSVP signaling.

In order to implement any QoS strategy using the QoS toolkit, the router and version of IOS must support the features used. Cisco provides a matrix of IOS versions, routers and QoS features for cross-reference [52].

Packet Classification and Marking

Packet classification occurs by marking packets using either IP Precedence or the DiffServ Code Point (DSCP) [46]. IP Precedence utilizes the three precedence bits in the IP version 4 header's Type of Service (ToS) field to specify class of service for each packet. Six classes of service may be specified. The remaining two classes are reserved. The DSCP replaces the ToS in IP version 6 and can be used to specify one of 64 classes for a packet. In a Cisco router, IP Precedence and DSCP packet marking can be performed explicitly through IOS commands or IOS features such as policy-based routing (PBR) and committed access rate (CAR) can be used for packet classification [53].

PBR [54] is implemented by the QoS Policy Manager (QPM) [51]. PBR allows for the classification of traffic based on access control list (ACL). ACLs [55] establish the match criteria and define how packets are to be classified. ACLs classify packets based on port number, source and destination address (e.g., all traffic between two sites) or Mac address. PBR also provides a mechanism for setting the IP Precedence or DSCP providing a network the ability to differentiate classes of service. PBR finally, provides a mechanism for routing packets through traffic-engineered paths. The Border Gateway Protocol (BGP) [56] is used to propagate policy information between routers. Policy propagation allows packet classification based on ACLs or router table source or destination address entry use with IP Precedence.

CAR [57] implements classification functions. CAR's classification service can be used to set the IP Precedence for packets entering the network. CAR provides the ability to classify and reclassify packets based on physical port, source or destination IP or MAC address, application port, or IP protocol type, as specified in the ACL.

Congestion Management

Congestion Management features are used to control congestion by queuing packets and scheduling their order of transmittal using priorities assigned to those packets under various schemes. Giroux and Ganti provide an overview of many of the classic approaches [58]. Cisco's IOS implements four queuing and scheduling schemes: first in first out (FIFO), weighted fair queuing (WFQ), custom queuing (CQ) and priority queuing (PQ). Each is described in the following subsections [53].

First In First Out (FIFO)

In FIFO, there is only one queue and all packets are treated equally and serviced in a first in first out fashion. FIFO is the default queuing mechanism for

above E1 (2.048 Mb/s) Cisco routers and is the fastest of Cisco's queuing and scheduling schemes.

Weighted Fair Queuing

WFQ provides flow-based classification to queues via source and destination address, protocol or port. The order of packet transmittal from a fair queue is determined by the virtual time of the delivery of the last bit of each arriving packet. Cisco's IOS implementation of WFQ allows the definition of up to 256 queues.

In IOS, if RSVP is used to establish the QoS, WFQ will allocate buffer space and schedule packets to guarantee bandwidth to meet RSVP reservations. RSVP is a signaling protocol, which will be discussed later in this section, the largest amount of data the router will keep in queue and minimum QoS to determine bandwidth reservation.

If RSVP is not used, WFQ, like CQ (see Custom Queuing), transmits a certain number of bytes from each queue. For each cycle through all the queues, WFQ *effectively* transmits a number of bytes equal to the precedence of the flow plus one. If no IP Precedence is set, all queues operate at the default precedence of zero (lowest) and the scheduler transmits packets (bytewise) equally from all queues. The router automatically calculates these weights. The weights can be explicitly defined through IOS commands.

Priority Queuing

In Cisco's IOS, PQ provides four queues with assigned priority: high, medium, normal, and low. Packets are classified into queues based on protocol, incoming interface, packet size, or ACL criteria. Scheduling is determined by absolute priority. All packets queued in a higher priority queue are transmitted before a lower priority queue is serviced. Normal priority is the default if no priority is set when packets are classified.

Custom Queuing

In Cisco's IOS, CQ is a queuing mechanism that provides a lower bound guarantee on bandwidth allocated to a queue. Up to 16 custom queues can be specified. Classification of packets destined for a queue is by interface or by protocol. CQ scheduling is weighted round robin. The weights are assigned as the minimum byte count to be transmitted from a queue in a given round robin cycle. When a queue is transmitting, the count of bytes transmitted is kept. Once a queue has transmitted its allocated number of bytes, the currently transmitting packet is completed and the next queue in sequence is serviced.

Traffic Policing and Shaping

Policing is a non-intrusive mechanism used by the router to ensure that the incoming traffic is conforming to the service level agreement (SLA). Traffic Shaping modifies the traffic characteristics to conform to the contracted SLA. Traffic shaping is fundamental for efficient use of network resources as it prevents the drastic actions the network can take on non-conforming traffic, which leads to retransmissions and

therefore inefficient use of network resources. The traffic shaping function implements either single or dual leaky bucket or virtual scheduling [59, 60, 61].

Signaling Mechanisms

End-to-end QoS requires that every element in the network path deliver its part of QoS, and all of these entities must be coordinated using QoS signaling. The IETF developed RSVP as a QoS signaling mechanism.

RSVP is the first significant industry-standard protocol for dynamically setting up end-to-end QoS across a heterogeneous network. RSVP, which runs over IP, allows an application to dynamically reserve network bandwidth by requesting a certain level of QoS for a data flow across a network. The Cisco IOS QoS implementation allows RSVP to be initiated within the network using configured proxy RSVP. RSVP requests the particular QoS, but it is up to the particular interface queuing mechanism, such as WFQ, to implement the reservation. If the required resources are available and the user is granted administrative access, the RSVP daemon sets arguments in the packet classifier and packet scheduler to obtain the desired QoS. The classifier determines the QoS class for each packet and the scheduler orders packet transmission to achieve the promised QoS for each stream. If either resource is unavailable or the user is denied administrative permission, the RSVP program returns an error notification to the application process that originated the request [62].

Study Results

The case study was intended to evaluate the feasibility and production implementation options for the

proposed DR solution. Our design sought to minimize communications requirements through data shadowing, exploitation of file commonality in file updates, network traffic shaping and to ensure system stability through IP QoS. Our prototype sought to measure the impact of each of the communication limiting techniques. The measurements were carried out in three distinct evaluations.

- The first evaluation was to determine the effect of block level updates versus file updates.
- The second evaluation was to determine the level of network bandwidth efficiency reasonably achievable.
- The third and final evaluation was establishing a configuration that supports the required QoS.

The test environment is presented next. Followed by the evaluations carried out and the issues they revealed. This section concludes with the results of the evaluations.

The Test Environment

A test environment was configured as shown in Diagram 3 and was constructed in as simple a manner as possible to reduce the cost of the evaluation. The entire test environment took about five days to install and configure, given that the infrastructure was already in place. Operating Systems and applications are loaded on the DR servers and updates are made manually. The baseline testing took about 90 days to gather the data. In the test environment, the OS was Solaris 7 and the applications were Oracle, PVCS and local configuration management database applications. This was the most labor-intensive part of the test set-up. One of the production servers was used to

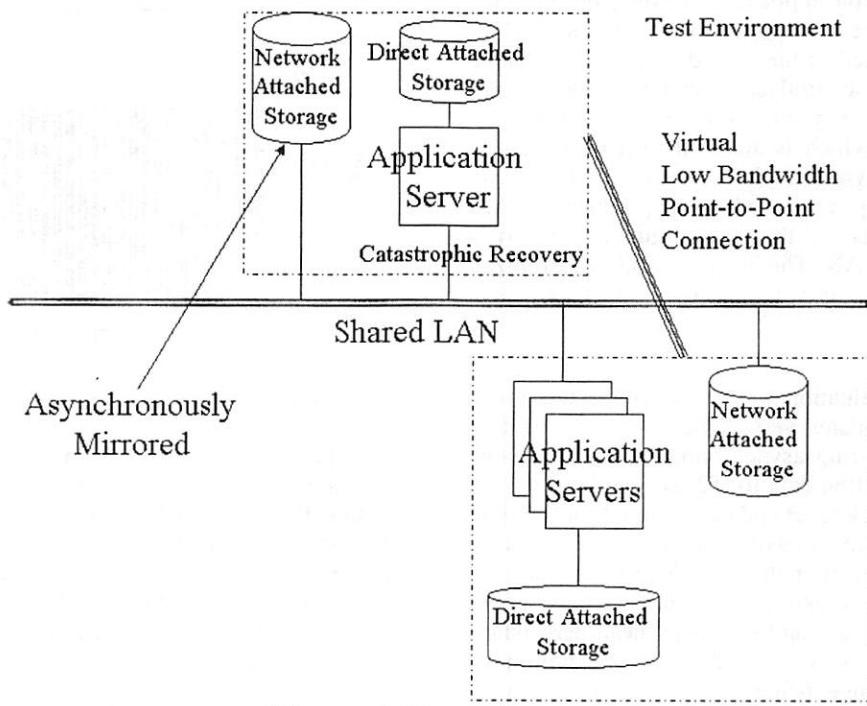


Diagram 3: Test environment.

maintain the heartbeat between the test systems; and initiate and monitor the asynchronous mirror updates. An available Network Appliance filer was setup to store the backup data. The primary data used in the test was restricted to NAS accessed data. This allowed the use of only one commercial data-shadowing product, reducing the cost and complexity of the test. The commercial data shadowing product chosen was SnapMirror [63] from Network Appliance. The filer and SnapMirror software were configured in approximately a day.

SnapMirror uses the snapshot [64] facility of the WAFL file system [65] to replicate a volume on a partner file server. SnapMirror is used in this study to identify changes in one of the production data volumes and resynchronize the remote volume with the active. After the mirror synchronization is complete, the backup data is at the state of the primary data at the instant of the snapshot. The remote mirrors are read-only volumes and are not accessible except by the source file servers. When a fail-over is to occur, the backup volumes are manually switched from a read-only standby state to read-write active state and rebooted. After the reboot, the backup filer is accessible by the DR server and the remote data can be mounted. SnapMirror and Network Appliance filers were chosen for this test based on their current use in the production environment, the availability of a filer to use in the test, and their support of block level updates allowing a determination of the impact of a block level verses file level update policy. The amount of data used in the test was constrained by the available disk storage on the filer, 120 GB.

The production application servers, the production NAS filers, the DR application server and the DR filer were connected to the shared production Gigabit Ethernet LAN. The production and DR filers along with the production application servers also have a second interface which is attached to a maintenance LAN running switched fast Ethernet. The asynchronous mirroring was tested on the production LAN to look for impacts and then reconfigured to run over the maintenance LAN. The heartbeat and synchronization initiation was carried out over the production LAN.

Evaluations

The first evaluation was to determine the effect of block level updates verses file updates. This test consisted of mirroring asynchronously approximately 100 GB of production data for 52 days and measuring the volume of block level updates required for the synchronization. The mirror synchronization was initiated daily at 3 pm, just after the peak load (see Chart 1). Chart 2 shows the weekday daily change rate as a percentage of total data monitored. The mean percentage daily rate of change was 2.32% with the minimum daily rate of change being 1.12% and a maximum daily change rate of 3.69%.

The sizes of the files that were modified over the update period were summed. This test was accomplished by running a perl script over the snapshot data used by SnapMirror. Block level updates show a reduction of approximately 50% of data required for transfer verses uncompressed copying of the modified files.

The second evaluation was to determine the level of network bandwidth efficiency reasonably achievable. The mirrored data was traffic shaped at the data source using a leaky bucket algorithm provided with the Snap-Mirror product. The data shadowing traffic was measured at the interface of the DR filer during the synchronization process. A threshold value (the hole in the bucket) of five megabits/second was set creating the virtual low bandwidth connection depicted in Diagram 3 within the LAN's Ethernet channel.

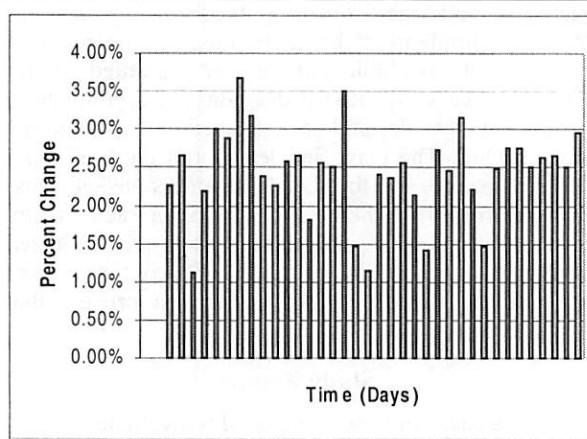


Chart 2: Daily data change rates.

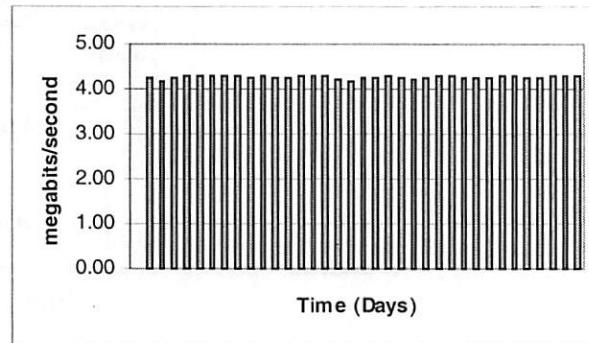


Chart 3: Weekday network throughput.

The asynchronous mirroring occurred over the production LAN where there is significant excess bandwidth. No other IP QoS mechanisms were used at this point in order to see if a constant load could be achieved and what impact the addition of this constant load would place on the Data Center filers and on the LAN. The rate of five megabits/second was selected as it was expected that a low bandwidth channel of less than five megabits/second would be required to update the production DR site given the current one

terabyte requirement. The mean effective throughput for the test was 3.99 megabits/second.

When looking at the data, it became obvious the overhead of determining the updates to the volume and read/write times were significant during periods of low data volume change which occur every weekend. In addition, the larger the data transferred in the update, the higher the network throughput. Removing data from each Saturday and Sunday yields an increase in the mean throughput to 4.26 megabits/second, which gives a bandwidth efficiency of 85%. Chart 3 graphs the weekday throughput of the test. As noted in chart 3, the throughput is consistent implying efficient network utilization. Throughput and peak network loads were measured before, during and after the synchronization on weekdays. Throughput, as expected, was increased proportionally to the network traffic added. Peak network loads were unaffected.

The second evaluation demonstrated two key capabilities. First, the data could be successfully synchronized between the primary and a remote DR site over a low bandwidth channel, in this case five Mbps. Secondly, the data required for this synchronization could be throttled to efficiently use a low bandwidth channel or provide a minimal impact on a shared higher bandwidth channel.

The third and final evaluation was establishing a configuration that supports the required QoS. The issue is to ensure that shaped data transmitted from the source filer is transmitted to the DR filer without additional delays. If the network cannot support the data rate of the source filer, the network acts as an additional queue and introduces delay. This delay introduces jitter into the shaped data that may prevent the synchronization of the data within the fidelity of the DR requirements. For example, if the DR requirement is to synchronize data hourly, the additional delay may cause the synchronization to take more than one hour and what is the result? Does the current synchronization fail and the next initiation of synchronization start, which could result in never successfully synchronizing the data? The proposed solution is two fold. First, initiation of a data synchronization does not occur until the completion of the previous data synchronization. If a synchronization has to wait, it is initiated as soon as the previous synchronization completes. This check was added to the heartbeat, but was also discovered to be a feature of the SnapMirror product.

Secondly, to prevent the additional delays in the network, IP QoS mechanisms can be used to provide a guarantee of adequate bandwidth based on the traffic shaping threshold. As previously described, many configuration options could be used to meet the QoS requirements. In the case study, the SnapMirror product was reconfigured to asynchronously mirror over the maintenance network. Custom queuing was

enabled on the interface to the source filer and configured to guarantee 5% of the 100 Mbps link to the mirror process. The maintenance network is primarily used for daily backup to tape, thus its traffic is bursty and heavily loaded during off-peak hours (8 pm-6 am). Network traffic continues to be measured at the interface of the DR filer and has continued to operate around the 4.26 Mbps level. An excerpt from the IOS configuration used in the test follows:

1. interface serial 0
2. custom-queue-list 3
3. queue-list 3 queue 1 byte-count 5000
4. queue-list 3 protocol ip 1 tcp 10566
5. queue-list 3 queue 2 byte-count 95000
6. queue-list 3 default 2.

Queues are cycled through sequentially in a round-robin fashion dequeuing the configured byte count from each queue. In the above excerpt, SnapMirror traffic (port 10566) is assigned to queue one. All other traffic is assigned to queue two. Entire packets are transmitted from queue one until the queue is empty or 5000 bytes have been transmitted. Then queue two is serviced until its queue is empty or 95000 bytes have been serviced. This configuration provides a minimum of 5% of the 100 Mbps link to the SnapMirror traffic.

Issues

Four issues arose during the proof-of-concept implementation. The first was ensuring data integrity. What happens if the communications line is lost, primary servers are lost, etc. during remote mirror synchronization?

The SnapMirror product was verified to ensure data integrity. Upon loss of network connectivity during a mirror resynchronization, the remote filer data remained as if the resynchronization has never began. However, an issue arose ensuring database integrity. This is a common problem when disk replication occurs via block level writes. The problem is the database was open during the snapshot operation so transactions continue. Furthermore, the redo logs were based from when the database last performed a "hot backup" or was restarted; thus, could not be applied to the backup. A solution to accomplish database synchronization is to actually shutdown the database long enough to take the snapshot. The database and the redo logs are then restarted. In the case study environment, the entire process takes about four minutes. Liu and Browning [34] provide details covering the backup and recovery process used. An alternative would be to purchase a disk replication package specific for the database.

The second issue was licensing. Several commercial products required for the DR functions use a licensing scheme based on the hostid of the primary system. The license key used for the primary installation could not be used for the backup installation and proper additional licenses had to be obtained and installed.

The third issue arose in the QoS testing and results from excess bandwidth and a reliance on production data to test the design. The maintenance network is lightly loaded during prime shift, from 6 am to 8 pm. The daily synchronization is initiated at 3 pm and normally completes in about 75 minutes. Since the maintenance network has excess bandwidth during the synchronization, it is possible that the custom queuing configuration has no effect as is indicated by the data.

This demonstrates the point that the configuration of bandwidth reservation through IP QoS mechanisms is only required on the local LAN when the local LAN does not have excess bandwidth capacity greater than that of the low bandwidth connection used for backup site communications. In most cases, the bandwidth reservation is insurance that the required bandwidth will always be available enabling the low-bandwidth link to be fully utilized. The issue of excess bandwidth in the testing environment is further evidenced when a synchronization was attempted without traffic shaping enabled. Intuitively, the peak load induced when a transfer of greater than two GB is initiated through a five Mbps queue would slow the transfer down. However, it did not as queue one's data continued to be serviced as long as there was no other network traffic.

The resolution of this testing issue is more difficult. In order to get valid test results for the quantity and types of changes, a production data volume were used. This requires non-intrusive testing on the production LANs. While testing on the maintenance network during backups would be useful to this project, it may also prevent production work from completing and has not currently been undertaken.

The final issue was security. Security of the remote servers, the remote mirror and communications is a topic, which must be further addressed. In the test, standard user authentication provides security on the remote servers and remote filers. Additionally, a configuration file, /etc/snapmirror.allow located on the primary filer, provides a security mechanism ensuring only the backup filers can replicate the volumes of the primary filers. The communication channels were over existing secure links. These secure links may not be available in the final target DR site.

Data Evaluation

The final task of the test is to determine the communication requirements to enable a stable geographically distributed storage update policy over a low bandwidth link. Since not all the data in the Data Center could be used in the prototype due to storage limitations, a linear estimation was used to predict the time required to perform the data synchronization. The assumptions of the model are:

- The amount of data changed is related to the amount of data in use.

- The amount of transferred data directly contributes to the length of time required for data synchronization.
- The relation between these two factors and the time required for data synchronization is linear.

Using the results from the sample data, 85% network efficiency allows a maximum of 13.84 GB of data to be transferred per 24 hours over a dedicated 1.544 Mbps T1 link. Under the assumption of the 3.69% maximum daily change rate, a maximum data store of 375 GB is supported by this design with a 24-hour synchronization policy. Under the assumption of the 2.32% mean daily change rate, a maximum data store of 596 GB can be supported. In order to support the required one TB, a minimum of a 4.22 Mbps link is required for the maximum daily change rate and a minimum of a 2.65 Mbps link is required for the mean daily change rate.

Summary and Future Work

This paper proposes a design that is the integration of several existing HA and network efficiency techniques, disk replication products and current IP QoS mechanisms, to establish an off-site DR facility over a low-bandwidth connection. The paper evaluates an approach to minimizing the communication requirements between the primary and backup site by relying on block level updates by the disk replication products to exploit file commonality in file updates; network traffic shaping and data shadowing to enable efficient network communications; and IP QoS mechanisms to insure that adequate bandwidth is available to ensure efficient usage of the low bandwidth link and that data synchronization can occur within the constraints of the DR requirements.

The proof-of-concept test developed for the case study demonstrated the functionality of the design over a reasonably low bandwidth connection of five Mbps and also demonstrated that a dedicated T1 link was insufficient given a 24-hour update cycle of one TB of data with the derived set of usage parameters. The proof-of-concept also demonstrated several other points about the design. First, the gain from the exploitation of file commonality can be significant but is of course usage dependent. In general, data replication products do not support block level updates and if they do, within a single product line. A more generic solution appears to be the exploitation of commonality at the file abstraction level where data compression and the integration of security mechanisms such as Internet X.509 [41] can be used for additional reductions in required bandwidth and increased security. Secondly, traffic shaping the data was demonstrated to be a highly effective method to efficiently use the available communications on low-bandwidth links. Finally, as stated in the previous section, the testing of the bandwidth guarantees is incomplete, difficult to measure and only required when excess bandwidth is

not available or more generally put, as an insurer of available bandwidth. Bandwidth reservations are most likely to be required when communications are over a heavily used or bursty WAN.

The next steps for this project take two distinct tracts. The first involves adding additional remote disk capacity, securing an appropriate remote link with a SLA of a minimum of five Mbps and testing additional disk replication products to support the full data set required at the DR site. The second is investigating the feasibility of providing an enhancement that offers support for asynchronous mirroring of only the modified areas of raw data in a compressed, secure manner, exploiting file commonality and further reducing bandwidth requirements. An enhancement or extension to the Network Data Management Protocol (NDMP) is being explored as a possible solution.

Author Info

Kevin Adams is a lead Scientist for the Submarine Launched Ballistic Missile program at the Dahlgren Division of the Naval Surface Warfare Center. He has been involved with Unix system administration since 1987. Kevin holds a B.S. in Computer Science from James Madison University and an M.S. in Computer Science and an M.S. in Electrical Engineering from Virginia Tech. Reach him via U. S. Mail at NSWCDD; Code K55; 17320 Dahlgren Road; Dahlgren VA 22448-5100. Reach him electronically at AdamsKP@nswc.navy.mil.

References

- [1] Gigabit Ethernet Alliance, 10GEA White Papers, Gigabit Ethernet:1000BaseT, http://www.10gea.org/GEA1000Baset1197_rev-wp.pdf, November 1997.
- [2] Alvarado, M. and P. Pandit, "How Convergence Will End the SAN/NAS Debate," *DM Review*, http://www.dmreview.com/editorial/dmreview/print_action.cfm?EdID=3016, February 2001.
- [3] Blackburn, D. and D. Driggs, *Sun Sets Agenda on Availability with new SunUP Program*, Press Release, Palo Alto, CA, <http://www.sun.com/smi/Press/sunflash/1999-02/sunflash.990209.3.html>, February 9, 1999.
- [4] Kleiman, S. R., S. Schoenthal, A. Row, S. H. Rodrigues, and A. Benjamin, "Using NUMA interconnects for highly available filers," *IEEE Micro*, Vol. 19, Issue 1, pp. 42-48, Jan-Feb 1999.
- [5] McDougall, R., "Availability – What It Means, Why It's Important, and How to Improve It," *Sun Blueprints Online*, <http://www.sun.com/solutions/blueprints/1099/availability.pdf>, October 1999.
- [6] Scott, D., J. Krischer, J. Rubin. *Disaster Recovery: Weighing Data Replication Alternatives*, Research Note T-13-6012, Gartner Research, June 15 2001.
- [7] Sun Microsystems White Paper, *Sun Ray 1 Enterprise Appliance Overview and Technical Brief*, August 1999.
- [8] Sun Microsystems Technical White Paper, *Sizing Sun Ray Enterprise Servers*, January 2000.
- [9] Sun Microsystems Technical White Paper, *Integrating Sun Ray 1 Enterprise Appliances and Microsoft Windows NT*, January 2000.
- [10] Wood, A., "Predicting client/server availability," *IEEE Computer*, Vol. 28 Issue 4, pp. 41-48, April 1995.
- [11] Barr, J. G., *Choosing a High-Availability Server*, Docid: 00018376, Faulkner Information Services, 2001.
- [12] Marcus, E. and H. Stern. *Blueprints for High Availability: Designing Resilient Distributed Systems*, pp. 5-6 & 298-300, John Wiley & Sons, Inc., 2000.
- [13] Patterson, D. A., G. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (RAID)," *Proceedings of the Conference on Management of Data*, pp. 109-116, Chicago, Illinois, June 01-03, 1988, United States.
- [14] Hiles, A., "Surviving a Computer Disaster," *IEEE Engineering Management Journal*, Vol. 2 Issue 6, pp. 271-274, Dec. 1992.
- [15] Powers, C. B., "Preparing for the worst," *IEEE Spectrum*, Vol. 33 Issue 12, pp. 49-54, Dec. 1996.
- [16] Jorden, E., "Project Prioritization and Selection: The Disaster Scenario," *HICSS-32, Proceedings of the 32nd Annual Hawaii International Conference on Systems Sciences*, 1999.
- [17] Kahane, Y., S. Neumann, and C. S. Tapier, "Computer Backup Pools, Disaster Recovery, and Default Risk," *CACM*, Vol. 31, No. 1, January 1988.
- [18] Gracia-Molina, H. and C. A. Polyzois, "Issues in Disaster Recovery," *IEEE Compcon Spring 1990, Intellectual Leverage, Digest of Papers, Thirty-Fifth IEEE Computer Society International Conference*, pp. 573-577, 1990.
- [19] King, R. P., N. Halim, H. Garcia-Molina, and C. A. Polyzois, "Overview of Disaster Recovery for Transaction Processing Systems," *Distributed Computing Systems, Proceedings, Tenth International Conference on*, pp. 286-293, 1990.
- [20] Wiesmann, M., F. Pedone, A. Schiper, B. Kemme, and G. Alonso, "Database Replication Techniques: A Three Parameter Classification," *Proceedings of the Nineteenth IEEE Symposium on Reliable Distributed Systems*, pp. 206-215, 2000.
- [21] Green, R. E., "Safety First," *IEEE Information Technology 1990, Next Decade in Information Technology, Proceedings of the Fifth Jerusalem Conference on (Cat. No. 90TH0326-9)*, pp. 593-595, 1990.

- [22] Schneider, F. B., "Implementing Fault Tolerant Services Using the State Machine Approach: A Tutorial," *Computing Surveys*, Vol. 22, Issue 4, pp. 299-319, December 1990.
- [23] Pfister, G. F., *In Search of Clusters: The Coming Battle in Lowly Parallel Computing*, Prentice Hall, 1995.
- [24] Laprie, J. C., J. Arlat, C. Beounes, and K. Kanoun, "Definition and Analysis of Hardware- and Software-Fault-Tolerant Architectures," *IEEE Computer*, Vol. 23 Issue 7, pp. 39-51, July 1990.
- [25] Milojicic, D. S., F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, "Process Migration," *ACM Computing Surveys*, Vol. 32, No. 3, pp. 241-299, September 2000. 241-299.
- [26] Alsberg, P. A. and J. D. Day, "A Principle for Resilient Sharing of Distributed Resources," *Proceedings of the Second International Conference of Software Engineering*, pp. 627-644, October 1976.
- [27] Wong, B., "Storage Area Networks: A Blueprint for Early Deployment," *Sun Microsystems Blueprints Online*, <http://www.sun.com/blueprints/0101/Storage.pdf>, January 2001.
- [28] Kovatch, M., "Fiber Channel-Arbitrated Loop: How Does It Compare With Serial Storage Architecture?" *NSWCDD White Paper*, <http://www.nswc.navy.mil/cosip/aug98/tech0898-2.shtml>.
- [29] Rodriguez, E., M. Rajagopal, and R. Weber, *Fibre Channel Over TCP/IP (FCIP)*, Internet Draft RFC, <http://www.ietf.org/internet-drafts/draft-ietf-ips-fcovertcpip-11.txt>, 06/19/2002 (expires December 2002).
- [30] Microsoft Corporation, *Common Internet File System*, <http://msdn.microsoft.com/workshop/networking/cifs>.
- [31] Sandberg, R., D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and Implementation of the Sun Network File System," *Proceedings of the Summer 1985 USENIX Conference*, pp. 119-130, Portland, OR, June 1985.
- [32] Hitz, D., J. Lau, and M. Malcolm, *File System Design for an NFS File Server Appliance*, Network Appliance, Inc., Technical Report (TR) 3002, http://www.netapp.com/tech_library/3002.html.
- [33] Rosenblum, M. and J. K. Ousterhout, "The Design and Implementation of a Log-Structured file system," *ACM Transactions on Computer Systems (TOCS)*, Vol. 10, No. 1, pp. 26-52, Feb. 1992.
- [34] Liu, J. and J. Browning, *Oracle7 for UNIX: Backup and Recovery Using a NetApp Filer*, Network Appliance, Inc., Technical Report (TR) 3036.
- [35] Bhide, A., E. N. Elnozahy, and S. P. Morgan, "A Highly Available Network File Serve," *USENIX Conference Proceedings*, USENIX, Dallas Texas, pp. 199-206, January 21-25, 1991.
- [36] Khalidi, Y. A., J. M. Bernabeu, V. Matena, K. Shirriff, and M. Thadani, *Solaris MC: A Multi-Computer OS*, Sun Microsystems Laboratories, SMLI TR-95-48, November 1995.
- [37] Vogels, W., D. Dumitriu, K. Birman, R. Gamache, M. Massa, R. Short, J. Vert, J. Barrera, and J. Gray, "The Design and Architecture of the Microsoft Cluster Service," *IEEE*, pp. 422-431, 1998.
- [38] Veritas White Paper, *Veritas Cluster Server v 2.0 Technical Overview*, http://eval.veritas.com/downloads/pro/vcs20_techover_final_0901.pdf, September 2001.
- [39] Brunt, B., F5 Networks White Paper, *Achieving High Availability: Quest Software's SharePlex & F5 Network's BIG-IP/3-DNS*, http://www.f5networks.com/solutions/whitepapers/WP_SharePlex_F5.pdf.
- [40] TurboLinux White Paper, *Cluster Server 6*, http://www.turbolinux.com/products/tcs/cluster6_whitepaper.pdf, October 2000.
- [41] Housley, R., W. Ford, W. Polk, and D. Solo, *Internet X.509 Public Key Infrastructure Certificate and CRL Profile*, RFC 2459, IETF, January 1999.
- [42] Dias, D. M., W. Kish, R. Mukherjee, and R. Tewari, "A Scalable and Highly Available Web Server," *IEEE Proceedings of COMPCON 1996*, pp. 85-92, 1996.
- [43] Iyengar, A., J. Challenger, D. Dias, and P. Dantzig, "High-Performance Web Site Design Techniques," *IEEE Internet Computing*, pp. 17-26, March-April 2000.
- [44] Cardellini, V., M. Colajanni, and P. S. Yu, "Geographic load balancing for scalable distributed Web systems," *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, Proceedings, Eighth International Symposium on*, pp. 20-27, 2000,
- [45] Bhatti, Saleem N. and Jon Crowcroft, "QoS-Sensitive Flows: Issues in IP Packet Handling," *IEEE Internet Computing*, pp. 48-57, July-August 2000.
- [46] G. Armitage, *Quality of Service in IP Networks*, pp. 67-70, 84-87, 105-112, Macmillan Technical Publishing, 2000,
- [47] Cisco IOS Documentation, *Quality of Service Solution Guide*, Implementing DiffServ for end-to-end Quality of Service, Release 12.2, pp. 371-392, http://www.cisco.com/univercd/cc/td/doc/product/software/ios122/122cgcr/fqos_c/fqcppt7/qcfdfsrv.pdf.
- [48] Braden, R., L. Zhang, S. Berson, S. Herzog, and S. Jamin, *Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification*, RFC 2205, September 1997.

- [49] Cisco Internet White Paper, *Designing Service Provider Core Networks to Deliver Real-Time Services*, http://www.cisco.com/warp/public/cc/pd/rt/12000/tech/ipra_wp.pdf, January 2001.
- [50] Cisco IOS Documentation, *Internetworking Technology Overview*, Chapter 49 Quality of Service Networking, Release 12.2, http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/qos.pdf.
- [51] Cisco IOS 12 Documentation, *Using QoS Policy Manager*, Chapter 1, Planning for Quality of Service, <http://www.cisco.com/univercd/cc/td/doc/product/rtrmgmt/qos/qpm21/qpm21ug/ugintro.pdf>.
- [52] Cisco Internet White Paper, *Supported Devices and QoS Techniques for IOS Software Releases*, <http://www.cisco.com/univercd/cc/td/doc/product/rtrmgmt/qos/pqpm20/qprodev.pdf>, July 2002.
- [53] Cisco IOS Documentation, *Quality of Service Solutions Configuration Guide*, Release 12.2, pp. 1-84, http://www.cisco.com/univercd/cc/td/doc/product/software/ios122/122cger/fqos_c/qcfbook.pdf.
- [54] Cisco Internet White Paper, *Policy-Based Routing*, http://www.cisco.com/warp/public/cc/techno/protocol/tech/plicy_wp.pdf, 1996.
- [55] Cisco Internet White Paper, *CiscoWorks2000 Access Control List Manager 1.4*, http://www.cisco.com/warp/public/cc/pd/wr2k/caclm/prodlt/aclm_ov.pdf, 2002.
- [56] Lougheed, K., Y. Rekhter, *A Border Gateway Protocol (BGP)*, RFC 1163, IETF, June 1990.
- [57] Cisco IOS Documentation, *Committed Access Rate*, Release 11.12, <http://www.cisco.com/univercd/cc/td/doc/product/software/ios111/cc111/car.pdf>.
- [58] Giroux, Natalie and Sudhakar Ganti, *Quality of Service in ATM Networks: State-of-the-Art Traffic Management*, Prentice Hall, pp. 48-59, 101-109, 246, 1999,
- [59] Butto, M., E. Cavalero, and A. Tonietti, "Effectiveness of the Leaky Bucket Policing Mechanism in ATM Networks," *IEEE Journal on Selected Areas of Communications*, Vol. 9, No. 3, April 1991.
- [60] Dittman, L., S. Jacobson, and K. Moth, "Flow Enforcement Algorithms for ATM Networks," *IEEE Journal on Selected Areas in Communications*, Vol. 9, No. 3, April 1991.
- [61] Gallasi, G., G. Rigolio, and L. Fratta, "ATM: Bandwidth Assignment and Enforcement Policies," *Proceedings of the IEEE Globecom*, paper 49.6, Dallas Texas, November 1989.
- [62] Cisco IOS Documentation, *Quality of Service Solutions Configuration Guide*, Release 12.1, Signaling Overview, pp. 243-260, http://www.cisco.com/univercd/cc/td/doc/product/software/ios122/122cger/fqos_c/fqcprt5/qcfsig.pdf.
- [63] Brown, K., J. Katcher, R. Walters, and A. Watson. *SnapMirror and SnapRestore: Advances in Snapshot Technology*, Network Appliance, Inc., Technical Report (TR) 3043.
- [64] Marchi, M. J. and A. Watson, *The Network Appliance Enterprise Storage Architecture: System and Data Availability*, Network Appliance, Inc., Technical Report (TR) 3065.
- [65] Hitz, D., *A Storage Networking Appliance*, Network Appliance, Inc., Technical Report (TR) 3001.

Embracing and Extending Windows 2000

Jon Finke – Rensselaer Polytechnic Institute

ABSTRACT

We were recently presented with the challenge of deploying a large scale Windows 2000 environment, initially for the Administration Division, but eventually including academic and other users. Rather than try to eventually re-integrate independently administered domains, we took this as an opportunity to develop the tools and resources to provide a campus-wide Windows 2000 environment that is well integrated with the existing enterprise information and computing systems, much like we integrated our Unix systems. This would automate many of the mundane administrative functions, yet provide appropriate delegation of control to departmental administrators as needed. This paper describes the systems we developed to make this happen.

Introduction

Rensselaer recently embarked on a major building initiative: a BioTech research center, an electronic media and performing arts center, a new central boiler and chiller plant, a parking garage, and a new campus entrance, and with this new construction come some new requirements for information sharing and archiving. The Administration division decided to handle this with a Windows 2000 Exchange email system. The good news is that they came to the Chief Information Officer to ask for help. The bad news is that the CIO agreed to help.

With this new initiative, we thought that it was very important to get this new system deployment right “the first time,” as attempting to go back later and fix things would be very difficult. We also wanted to look beyond the requirements of this specific project, and deploy a campus-wide solution to integrate support of academic programs as well as other administrative units. Our existing Windows administrators were spread over a number of administrative units, as well as a few academic departments. It was important to come up with a system that they would be comfortable with, and one where they would be willing to “turn over control” to the central computing center. This was quite a change in approach for our Windows administrators.¹

At Rensselaer, we have a long history of automating our Unix systems administration tasks (via an Oracle database) and striving to get information from an authoritative source. Figure 1 gives a high level view of the flow of people-related information at Rensselaer. Human Resources and student record data flows from the administrative system running SCT/Banner into the Simon system’s [4, 5] Oracle database. One of the things this is used for, is to automatically create and expire Unix/email user accounts. From Simon, the relevant information is sent to each of the various client information and authentication

systems, including the central telephone directory (LDAP, ph and web based), the ID card system, AFS and Kerberos, and the new Windows 2000 domain and Exchange email system.

In addition, we have developed many tools and techniques that allow us to delegate responsibility with a great deal of fine-grained control. For example, our telephone directory system [6] allows a person from each department to maintain directory information for their staff. Our eventual proposal was actually built on top of several existing systems, with some enhancements and extensions.

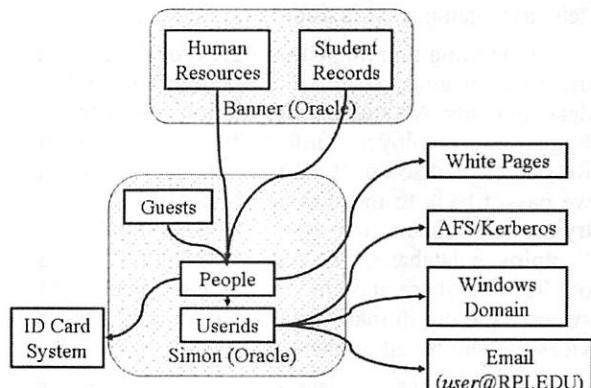


Figure 1: Information flow at Rensselaer.

One of our existing projects was to provide a comprehensive LDAP [9] directory service, and make this the directory service of choice for information consumers. To this end, the data needs to be both accurate as well as timely. Since it appeared that our LDAP service would be driving our Active Directory service, this project gained a key role in our Windows 2000 project.

Another existing project was single signon, or at least some form of password synchronization between systems. While we may have a bunch of different systems behind the scenes, our users think that they have a single account and password and we provide the magic to make it all work. To this end, we wrote a

¹During some training on MS Exchange, we learned that Microsoft invented Kerberos.

web-based password changing tool. Unlike the one developed at Auburn University [10], we use public key encryption to protect and store the passwords; and we use a relational database to manage and store both the public keys and the encrypted passwords, as well as to drive the actual password changing process on Windows, Oracle and Kerberos.

In our original Windows 2000 proposal, the organization hierarchy would be based on the University structure, as defined in the telephone directory. However, after meeting with some of the Windows administrators, we saw that we needed a way to allow these folks to create their own structure, yet keep them from interfering with other administrators. Fortunately, our existing telephone directory model provided a good fit in both respects, and we decided to extend it to handle the Windows domain.

The Institutional Layer

While there are many technical challenges to implementing a system like this, institutional politics add a number of other “opportunities to excel.” In our case, we were fortunate that we had addressed many of these in the past, having long established the practice of feeding data from Human Resources and the Registrar into the Simon system to automatically create Unix and email accounts for everyone on campus. This same system was grown to manage the campus telephone directory and feed the campus ID card system. With these projects in place, we had established our credibility in delivering campus-wide information services.

One thing that helped us a great deal was always insisting on going to the authoritative source for all data elements. All student information comes from the Registrar, employee information from Human Resources, and so on. If we have changes to that data, we pass it back to those systems. Because we are not trying to maintain our own “student database” or “employee database,” we can insist that the “owners” of the data share it with us. We have been able to sweeten the pot in many cases by providing some services to them based on their data.

As information systems evolve, we sometimes move away from this ideal because user demands may outpace the ability of support organizations to react to changes. As a result, we have discovered that several important data elements that should have been maintained centrally had moved into the Simon system.² We now have an ongoing task force monitoring the relationship between Simon and the rest of the administrative information systems.

Care and Feeding of LDAP

We begin by describing the development of our existing LDAP-related information flow, which provides

²If you are publishing the enterprise phone directory, you WILL get good data feeds, and the rest of your projects can benefit.

the base upon which the Windows implementation is built. We needed a way to detect changes to data in our enterprise information systems and propagate just those changed records to LDAP. Since we wanted these changes to be as close to “real time” as possible, the process needed to be very lightweight. By making the load on the enterprise database³ very small, we can make frequent checks for changes without annoying the DBAs or impacting performance on that system.

To make things more interesting, we had to do this in such a way to make a minimal impact on the keepers of the enterprise database. These MIS folks have a great many demands on their time, and are often not available to take a large role in the deployment of new services. This necessitated that our approach require very little involvement of the DBAs and developers in the MIS department.⁴

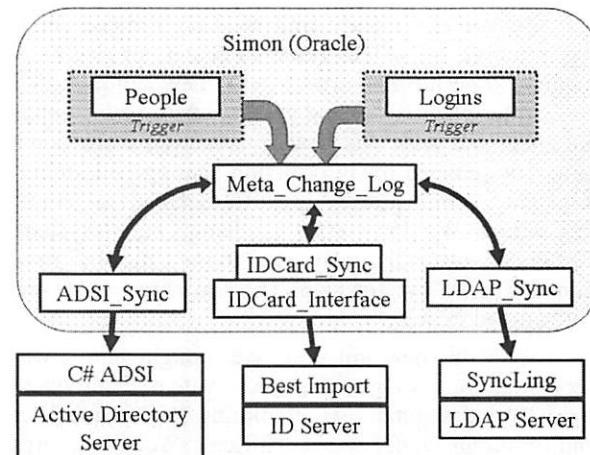


Figure 2: Information change flow architecture.

In our initial LDAP deployment, we were populating a general “people” directory following the “eduPerson”⁵ [3] to replace our PH white pages directory server. A second LDAP project was to provide a “POSIX” account space to provide /etc/passwd to our Unix hosts. We did this by generating LDIF files using a PL/SQL package [13]⁶ and some interface code to generate these into files [7]. While generating LDIF files to test the LDAP servers was okay, this was not going to handle the ongoing updates that we needed.

Triggers and Change Queues

A powerful feature of Oracle and other high end databases is the ability to define triggers [1] that will

³We run SCT/Banner for our student records, finance, and payroll.

⁴While this is a local condition, I suspect over-booked MIS shops are a fact of life at many other sites.

⁵This is an effort by EDUCAUSE to come up with a set of standard attributes for a directory entries at educational sites.

⁶PL/SQL Packages are a collection of functions, procedures, cursors, and variables. These can be both public and private. Access to the public procedures can be granted to Oracle users or roles. Once accessed in a session, a package maintains state between calls.

automatically execute when a particular table is changed in some way. This gives us a way to insert our own business rules in the general operation of the database. For example, our code will get executed when a person is added to the people table. What we did is record key information about the change into a queue table that would be processed later. Our intention here was to place some of these triggers into our main administrative database, SCT/Banner. In this way, we would not need to change the vendor code, which should make handling new releases much easier. For applications we write ourselves, we have the option of using triggers or putting these calls directly in the applications themselves.

In Figure 2, we have triggers “watching” for changes to the Logins and the People tables. When a change in a table is detected, we write a record into the Meta_Change_Log table (see Table 1). We include a number of identifying fields. The first three – Tname, Subtype and Change_Type – identify which table, and in some cases, which part of the table was changed and how it was changed. We also have a number of fields to identify which entry was changed. Since many of the changes we are interested in involve people, we include the primary keys used by each of our two main information systems. However, we may be looking for changes to something other than people

(department names, group membership, etc.) so we have a generic character and numeric key available for tables dealing with non “people” information. The exact set of identifiers used depends on the table.

The second half of the Meta_Change_Log (see Table 2) deals with how we process these records and manage the queue. Each entry has an entry date and a sequence number to assist with ordering. We are feeding several systems with changes: our LDAP directory server, our Windows 2000 Active Directory Server and our Photo ID Card system. We need to be able to process each of these queues independently, since they may be operating with different schedules for backup, maintenance, etc. All of these queues has a “Process Needed” flag and a process date. When a record is inserted into the queue, the appropriate “Process Needed” flags are set to “Y”.

Each of the _Proc flags has an index on it. When a record is processed, this flag is set to Null. Oracle do not index Null values, so that only the active (change pending) records are included in the index. This means that, in normal operation, these indexes are very small and can be accessed and updated very quickly. This keeps the load on the database to a minimum and allows us to make frequent checks to look for changed records.

Name	Type	Size	Description
Tname	Varchar2	32	Identifies the table that was changed
Subtype	Varchar2	32	Optional subtype to allow specialized processing based on which fields in a table were changed.
Rrowid	Rowid		The Rowid (Oracle record identifier) of the row that was changed. This allows for direct access to the desired row with no searching needed.
Change_Type	varchar2	1	A flag indicating Insert, Modify or Delete of the record.
PIDM	Number		Person identifier for our administrative system (SCT/Banner)
Person_Id	Number		Person identifier for the Simon system.
Pkey_String	Varchar2	32	A table specific character string to identify the record in question (such as a Unix account name).
Pkey_Number	Number		A table specific numeric value to identify the record in question (such as a Unix UID).

Table 1: Meta_Change_Log Table – Identification.

Name	Type	Size	Description
Entry_Date	Date		The time and date when the record was entered.
Entry_Number	Number		An ever increasing sequence number.
LDAP_Proc_Date	Date		The date when the LDAP SyncLing processed this record.
LDAP_Proc	Varchar2	1	A flag set to “Y” when this record is awaiting processing by LDAP.
ADSI_Proc_Date	Date		The date when the ADSI synch program processed this record.
ADSI_Proc	Varchar2	1	A flag set to “Y” when this record is awaiting processing by ADSI.
IDCARD_Proc_Date	Date		The date when the IDCARD synch program processed this record.
IDCARD_Proc	Varchar2	1	A flag set to “Y” when this record is awaiting processing by IDCARD.

Table 2: Meta_Change_Log Table – Queue processing.

SyncLings

Now that we had a list of changes that needed processing, we also needed a way to get them into LDAP. To this end, we wrote an Oracle package called `LDAP_Sync`. This package references the `Meta_Change_Log` table and provides two procedures, `Get_Changes` and `Ack_Change`. The `LDAP_SYNC` package is called by a Java program called a “SyncLing.” It calls the `Get_Changes` routine to get the next change, processes it, and then marks that change as done with the `Ack_Change` routine.

The `Get_Changes` routine (see Table 3) is called, and the next LDIF record [8]⁷ is returned. This is then compared to the existing record in the LDAP server by the SyncLing, and the appropriate action is taken. As each record is processed, the `Ack_Change` routine is called with the `Rec_Id`. This will mark the record as processed (clearing the `LDAP_Proc` flag in the table). This process is repeated until `Get_Changes` returns a null record. At this point, the SyncLing pauses for a few seconds, and then resume asking.

As part of the process of the initial data load of the LDAP server, we had already written a package to generate LDIF records for each person at Rensselaer from the directory database, so it was a trivial matter to call this routine from the `Get_Changes` routine.

We are actually populating several “trees” in our LDAP database: a general people tree, and a POSIX account tree (in part, to replace our `/etc/passwd` file and some group files). There are corresponding routines to generate flat LDIF files that we can call for each of these cases. The `type` field identifies the type of LDIF record being returned, so that the processing program knows what to do. It can also ask for just records of a given type, or it can ask for all types of records.

This system works very well for our Unix systems for which it is deployed. In the next section, we discuss how we extended it to service our Windows 2000 systems.

Feeding of Active Directory

Our initial plan was to use our LDAP server to feed the Active Directory⁸ server. However, this ran into a few problems. Although we were feeding both a

⁷One method of loading data into an LDAP server is using a format known as the “LDAP Data Interchange Format” or LDIF.

⁸Active Directory is the database server used by Windows 2000 to hold user information and passwords.

POSIX user id base and the general person/directory information into LDAP in essentially real time, the data did not match well with the requirements of Active Directory and our Exchange server. Although we were able to load people into Active Directory via LDAP, we ran into limitations due to our focus on the “eduPerson”; fields we needed in Active Directory were not available.

The second problem is that we needed to propagate password changes from our general system into the Windows 2000 world. Our password changing scheme (see section below) relies on public key encryption to secure the passwords while in transit; implementing this via LDAP was not practical.

We had recognized early in the project that LDAP was not going to be able to handle the password changing, so we had started a second project to manage the passwords. At that time, Microsoft was moving to the use of JAVA in some aspects of their systems, so we started development of a JAVA program that would talk to the database; get the password changes, and then apply them to Active Directory. When it became clear that our LDAP approach was not going to handle our needs, we expanded the role of the password propagation to a more general Active Directory update system.

The database side of the LDAP solution described above was exactly what we needed, so we simply cloned it. In fact used the same table, and simply added an ADSI (Active Directory Service Interface) [11] propagation flag to duplicate the LDAP propagation flag, and duplicated the PL/SQL package to provide ADSI with its own interface to the database. This allows for ADSI and LDAP to run in parallel.

The `ADSI_Sync` package starts out much like the `LDAP_Sync` package with a `Get_Changes` and `Ack_Change` routine, and they operate in the same way as described above. However, rather than returning an LDIF record, we just return the username. We have also added two more routines, `Get_Dirinfo` (see Table 4) and `Get_Dirinfo2` (see Table 5). The rollout of this service was under some pretty tight time pressure, which is why these are two separate routines, rather than just one.

As the Windows 2000 service was being rolled out, and more administrative users were being added to our exchange server, some other issues were discovered. We were already using the Windows 2000 user base and password for students to access our

Name	Type		Description
RType	Varchar2	In/Out	On call, specifies the type of records desired, and on return, indicates the type of record being returned.
Rec_Id	Varchar2	Out	Returns a record id used for the <code>Ack_Changes</code> routine.
LDIF_Record	Varchar2	Out	An LDIF formatted record with the information for the next person to be processed.

Table 3: `Get_Changes` procedure parameters.

public workstation labs, so the password changing system was being well exercised, but as administrative users were moving their email from our Unix-based pop mail service to the Exchange server, we needed to feed more information into Active Directory.

One of the other objectives of this project was to provide Exchange based mailing lists for all the people in a given department or division. Our first pass at this was to provide a pair of routines, Get_Divisions and Username_By_Division; the first would return a list of all of the divisions, and the second would return a list of everyone in the specified division. In the same way, we also provided Get_Departments and Username_By_Department routines. These would be used to maintain the desired membership lists. However, we quickly ran into a problem with the names being used; different aspects of the Active Directory service took exception to some of the special characters we were using such as ampersand, dash, and a few others. To handle this, we added a CLEAN flag to the Get_D... procedures and a Get_Username_By_Clean_D... call.

Once our users had tasted the wonders of automatically maintained mailing lists, they were hungry for more. As part of a different project, we had put together some special mailing lists for our ListProc machine such as "Deans, Directors and Department Heads" and "Building Coordinators." These lists were being dumped as flat files using our Generate_File program. A little bit of creative coding, and these lists became available to ADSI via the Get_Specials and Get_Username_By_Special.

To be Sharp, You Must C Sharp

When we first started working on a JAVA program to change passwords, it appeared that Microsoft would provide (JAVA) Class libraries which exposed

their ADSI routines. As it turns out, they ended up abandoning their JAVA efforts, and we were forced to provide our own JAVA callable ADSI routines by using JNI (Java Native Interface) to wrap a Windows DLL written in C. Although it worked, it was very difficult to change, and our development staff dreaded new requirements from the Windows team. Every new function would take 30 minutes to write, and then four days to debug and tweak so it would actually work.

Microsoft's new direction was a programming language called C# [12], part of their Visual Studio package. With C#, Microsoft has provided the ADSI class libraries we need, allowing us to eliminate the need for a custom DLL. This has proven much easier to work with, and we have resumed adding new fields and streams into Active Directory from our central database. This shows up as the C# ADSI box in Figure 2.

Making Changes to Passwords; Here, There and Everywhere

When we first rolled out our campus-wide Unix service, built on top of an AFS filestore, we wrote a replacement for the general Unix passwd program that would update the Kerberos password and save a copy of the Unix PASSWORD crypt in our central database. This enabled us to build conventional /etc/passwd files for a few legacy systems that did not use Kerberos (this was 10 years ago).

As our systems evolved, many of our users moved away from the Unix workstations for their computing, but continued to use their Kerberos passwords for email, printing, dial-up and other services. Instructing people to connect to a Unix machine (using ssh, not telnet!) sign on, and invoke the passwd program to change their password resulted in a lot of frustration for both our users and our help desk staff.

Name	Type		Description
Uname	Varchar2	In	Target username – provided by Get_Changes or other routines.
Pref_Email	Varchar2	Out	Preferred email address.
Camp_Phone	Varchar2	Out	Campus telephone number.
Camp_Fax	Varchar2	Out	Campus fax number.
Camp_Address	Varchar2	Out	Campus address.
Department	Varchar2	Out	Department Name.
Division	Varchar2	Out	Division Name.
Web_Page	Varchar2	Out	URL of person's web page, if available.
Title	Varchar2	Out	Title of person (employees only – not students.)

Table 4: Get_Dirinfo procedure parameters.

Name	Type		Description
Uname	Varchar2	In	Target username – provided by Get_Changes or other routines.
Last_Name	Varchar2	Out	Person's last name.
First_Names	Varchar2	Out	Person's first and middle names.
Preferred_First_Name	Varchar2	Out	Alternate first name preferred by the person. Used in the directory.

Table 5: Get_Dirinfo2 procedure parameters.

A web based password changing system was an obvious solution.

Although we had long ago stopped collecting a Unix crypt, we still liked the idea of collecting passwords for use on other systems, such as a trouble ticketing system (Oracle based), our academic Oracle servers, and our Kerberos 5 server. We were not comfortable with the idea of storing, or even transmitting, clear text passwords. While we could protect the web session with SSL, we still had the traffic from the web server to the database server. So we decided to use public key encryption, encrypt the plain text password with a public key on the web server, and transmit the encrypted text to the database for processing and storage. Since we were already connecting to the database for other reasons, this was a logical spot to store the public keys. As part of this process, the password change web page also signals (via Oracle signals) [2] the back end password processing.

Using Oracle to broker the password changes makes it much easier to add new authentication services. The web page used by the users does not need to change, nor does it need to understand every new type of authentication system. To add a new authentication system, we just need to write a new back end that understands that world, along with some public key encryption and database access. It also allows for people to request password changes when some authentication services are not available; the change is held until the service is restored.

In Figure 3, we have data flow for a password change. We start with a web page, connecting to our secure web server via an SSL connection (step 1). The secure web server does some password strength checks, and, if things are okay, makes an immediate password change to our AFS Kerberos server (step K).

There is also an option to simply test a new password to ensure that it will pass the strength tests.

Once the password change is checked out, the web server calls an packaged routine in the database, Get_Public_Key (step 2). This public key is used to encrypt the clear text of the user's new password, and then that encrypted text is stored back in the database (step 3) via the Store_Pw procedure. This stores the encrypted text, the user name, and the key number in the Encrypted_Passwd_Cache table (Table 6).

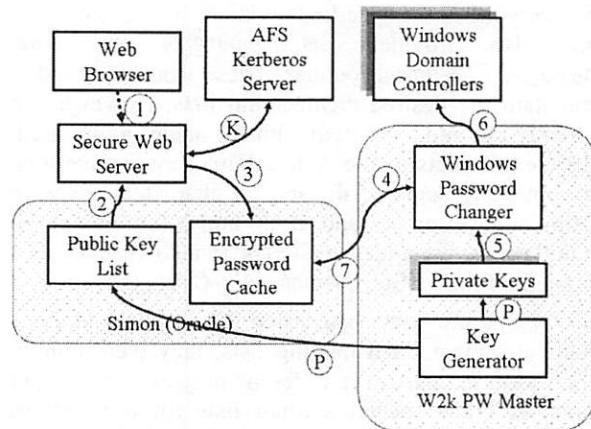


Figure 3: Password change flow.

Using the same sort of propagation polling that we used for the ADSI_Sync process, we have Windows Password Changer program running on our Windows 2000 Password Master machine. It looks for entries in the Encrypted_Passwd_Cache table (step 4) that have Windows_Prop_Pending set to "Y." Along with the principal name, and the encrypted password, it also gets a key number. It then consults its own list of private keys (step 5) and uses that to decrypt the

Name	Type	Size	Description
Principal_Name	Varchar2	32	The user name
Encrypted_Key	Varchar2	1024	The encrypted user password.
Key_Num	Number		The key number used to encrypt the password.
Entry_Date	Date		The time and date of the change request.
Simon_Prop_Pending	Varchar2	1	A flag indicating that this password needs to go to Simon.
Simon_Prop_Date	Date		The time and date when this change was given to Simon.
Simon_Err_Code	Number		A numeric error code from this operation.
Applix_Prop_Pending	Varchar2	1	A flag indicating that this password needs to go to Applix.
Applix_Prop_Date	Date		The time and date when this change was given to Applix.
Applix_Err_Code	Number		A numeric error code from this operation.
Kerb5_Prop_Pending	Varchar2	1	A flag indicating that this password needs to go to Kerberos version 5 server.
Kerb5_Prop_Date	Date		The time and date when this change was given to the Kerberos version 5 server
Kerb5_Err_Code	Number		A numeric error code from this operation.
Windows_Prop_Pending	Varchar2	1	A flag indicating that this password needs to go to Windows.
Windows_Prop_Date	Date		The time and date when this change was given to Windows.
Windows_Err_Code	Number		A numeric error code from this operation.

Table 6: Encrypted_Passwd_Cache table.

password and make the change on the Windows 2000 domain controller (step 6). Once that has been done, it then updates the Windows_Prop_Pending flag (step 7), as well as the Windows_Prop_Date field. It continues getting more rows to process until there are no more to be done. At that point, it will "sleep" for 30 seconds, and then check again. While this seems pretty quick, there is some delay in propagating the password changes between the Windows Domain Controllers, so it can take up to 15 minutes for the changes to get passed among the Windows servers. We have not found any ways in which to improve this.

Key Generation

There are a number of options and approaches to managing the public/private key pairs. For our Windows 2000 world, we have a key generator program that we run on the Windows 2000 Password Master machine which generates a key pair. The private key is stored in a secure location on the Password Master machine (and yes, we really need to keep this machine secure), and the public key is stored on the Oracle database server. This makes it available to the Secure Web Server when needed. The public keys are stored in the Passwd_Key_List table (Table 7).

We have not spent a lot of time working on our key management procedures, but we periodically generate new key pairs and remove the old private keys from service. The use of key numbers and types help keep things in order for us. However, since all of the applications (the secure web server and the back end processors) all rely on the Oracle procedures for key management, we can change those routines and the external code will do the right thing without any changes.

Multiple Streams

In the case discussed above, we are keeping two passwords in sync: our AFS Kerberos password and the one for our Windows 2000 server. In actuality, we are also keeping the Oracle password on the Simon server, as well as the Oracle password on our trouble ticketing system (Applix). We are maintaining our Kerberos version 5 server passwords via this approach.

When the secure web server encrypts a password, it actually does it twice, once with a key used

for the Windows Password Changer, and a second time using a different key pair used by the Simon system. So, each password change results in two entries being made in the Encrypted_Passwd_Cache table. When the Windows key is used, the Windows_Prop_Pending flag is set, and when the Simon key is used, the Simon_Prop_Pending, Applix_Prop_Pending and Kerb5_Prop_Pending flags are set. Since these three queues are all processed on the Simon server machine, they are able to share the same public and private keys.

At present, our Kerberos 5 server is not yet in production. By using its own propagation flag, changes to other servers can still go through when the Kerberos 5 server is not available. When service is restored, it can catch up on the changes. We also have the ability to re-encrypt the clear text to feed to other systems. This allows us to keep most of the processing on the back end server, rather than making additional encryption runs on the secure web server.

Timing Issues

Although we would like all of this to take place instantly, the back end program is doing decryption and re-encryption, as well as accessing the database. In addition, the program is actually run by another job scheduling system that is sometimes busy with other functions (creating accounts, changing quotas, etc.). We also have the problem that the system we are trying to update is sometimes down or unreachable, so the password change is stuck waiting in the queue. This was starting to result in some operational problems where a person would change their password, and then immediately try to access a service and get an authentication failure.

To assist our help desk (and advanced users), we wrote another web page which can display the exact date and time of a person's last password change, and when it was applied to each authentication base. It will also display if there is a change pending. This also lets us generate some statistics on the time it takes for password changes to propagate. The page will first report on any pending changes, and then report on the most recent set of completed changes. It produces a report like Table 8.

Name	Type	Size	Description
Key_Num	Number		The key number.
PK_N	Varchar2	2048	Part of the public key.
PK_E	Varchar2	32	The other part of the public key.
Active	Varchar2	4	A flag indicating that this key is the active key for this type.
Start_Date	Date		When a key goes into service.
End_Date	Date		When a key is removed from service.
Create_Date	Date		When a key was created.
Key_Type	Varchar2	4	What "TYPE" of key this – used to identify who has the private key that matches this key.

Table 7: Passwd_Key_List table.

There are a couple of oddball entries. For example, the Kerb 4 change always has an elapsed time of 0 seconds: if it fails, none of the Oracle processing will take place, and no logs will be written for display. Of course, the user will get a very clear error message on the web page when this happens. The other odd case is the AdminReq. This is a case where the user lost their password and went to the help desk and asked that their password be reset. This system uses a similar mechanism to the ones discussed here, and sometimes it is subject to delays.

Other password issues

For long term storage of “clear-text” passwords, in order to let us populate services yet to be deployed, we have the option of storing the appropriate private keys on secure storage (such as a floppy disk locked up in safe), and when we need to load the initial population of a new service, we get the floppy, restore the private keys, and load the new system.

Systems Administration as a White Pages problem

At LISA X, I presented a paper [6] showing how managing the University white pages (phone directory) was really a Systems Administration problem. Things have now come full circle, and we are applying our telephone directory tools and techniques to managing our Windows 2000 domains.

For our telephone directory, we take a data feed from Human Resources, decide (based on status and department) if they are included in the directory, or if they need to be moved to another part of the directory tree.⁹ We also have facilities available to manually move someone to a different department, have someone appear in a second or any number of departments. What is more, our tools allow us to delegate control of any department or subtree to folks outside of the computer center.

This is the same problem we wanted to solve with Windows 2000 user accounts. We wanted them to be created and expired automatically, we wanted new accounts to be put in the appropriate domains by default, and we wanted to allow our Windows 2000 administrators to have some ability to shuffle folks around, but not give every admin the ability to move

⁹Our baseline organization structure is based on our financial accounting system. We modify it slightly to reflect the actual organizational structure.

every account. The tools used for our telephone directory could be used for this with almost no changes.

Futures

Having the common Windows/Unix/Kerberos account and password base is very nice, in that it allows us to deploy new services requiring authentication and have some options with how we do it. Since all of the IDs and passwords are the same, the users do not know or even care how it is done. At present, we require the users to change their password in order to access Windows 2000 and some other services. We like this from a security perspective, as the initial password is stored in clear text and may have been printed. On the other hand, this creates an extra step for new users. However, it appears that user convenience has won out over security, and we will soon be removing the requirement to change the initial password.

We do have a web based account pickup system in place that does put new users right on the password change web page once they get their initial password. Many of the new users take that opportunity to change their passwords. Of the 1596 new users who used the web pickup tool, 1467 (92%) changed their passwords.

We will continue to add new fields into Active Directory from our enterprise information systems. Now that the tools are in place and understood, it is much easier add new things.

Problems and Lessons

One problem is our general job queuing mechanism, so we will be investigating adding a special password change thread or simply multiple threads to keep password changes from getting stuck behind other jobs. A priority setting for jobs might be enough, since, in general, any given job is pretty quick. The problem comes when a password change gets stuck behind 1200 user account creation requests the problems arise.

Working with public key encryption can be very tricky and we have run into problems with how keys are generated and stored. We have two different ways of generating key pairs, and although both store the public keys in Oracle, and we can successfully encrypt with either key, the private keys were not interchangeable. For example, if I generated a private key under AIX, it would not work in Windows. We have subsequently learned why this was, and have fixed this. We

Name	Requested At	Done At	Elapsed Time
AdminReq	09:47:07 Jun-10-2002	09:42:13 Jun-10-2002	00:00:06
Applix	10:43:34 Apr-23-2002	10:44:42 Apr-23-2002	00:01:08
Kerb 4	10:43:34 Apr-23-2002	10:43:34 Apr-23-2002	00:00:00
Kerb 5	10:43:34 Apr-23-2002	10:45:18 Apr-23-2002	00:01:44
Simon	10:43:34 Apr-23-2002	10:43:59 Apr-23-2002	00:00:25
Win2000	10:43:34 Apr-23-2002	10:43:53 Apr-23-2002	00:00:19

Table 8: Password status sample.

have also hit problems with packages working under one version of AIX on a particular type of processor, and, when recompiled on a different machine, not working at all. Some of the packages we are using appear to have some buffer overruns and other memory layout issues.

A Big Hammer

The requirement for a high end database server and some of the other infrastructure may make this solution "too big" for some small site to contemplate. However, I feel that part of this is a matter of perception. When we first embarked on the Simon project, using Oracle added a big expense for software and hardware, and a lot of development effort. On the other hand, the challenge of trying to deliver an enterprise wide computing environment was simply too big to accomplish without using commercial support. This project will not help a small operation; it is simply too big a hammer. But if you are working with tens of thousands of user accounts, and a large turnover in population, you have no other choice but to go for the big tools.

While the Simon system is involved in many facets of our information systems, one of the most important roles it plays is as an interface layer between different vendor applications. It gives a place to apply our business rules and needs to the areas where the vendor products fall short. Part of the cost of doing this is having a development staff who can make these systems work together.

While it is certainly nice to think that we can do everything with free software, I do not feel that that is realistic approach to a large scale system. At our site, we spend close to one million dollars a year in software licensing and support contracts. Software license costs are simply a cost of doing business. We don't seem to have a problem obtaining four servers for our LDAP directories, so a database server shouldn't be any different.

References and Availability

All source code for the Simon system is available on the web. See <http://www.rpi.edu/campus/rpi/simon/README.simon> for details. In addition, all of the Oracle table definitions as well as PL/SQL package source are available at <http://www.rpi.edu/campus/rpi/simon/misc/Tables/simon.Index.html>. We can make the ADSI (C#) routines available as well. It seems unlikely that we can distribute the public key encryption routines used in the password management, but presumably you can find them elsewhere.

Acknowledgments

I would like to thank AEleen Frisch for her shepherding of this paper, as well as Deb Wentorf of Communications and Collaboration Technologies at Rensselaer for her proofreading and editing. I also want to thank Rob Kolstad for his excellent (as usual) job of

typesetting this paper. Thanks also to Mike Douglass <douglm@rpi.edu> who wrote the LDAP part of the project, and Alan Powell <powela@rpi.edu> who did the ADSI part of the project.

Author Biography

Jon Finke graduated from Rensselaer in 1983, where he had provided microcomputer support and communications programming, with a BS-ECSE. He continued as a full time staff member in the computer center. From PC communications, he moved into mainframe communications and networking, and then on to Unix support, including a stint in the Nysernet Network Information Center. A charter member of the Workstation Support Group he took over printing development and support and later inherited the Simon project, which has been his primary focus for the past 11 years. He is currently a Senior Systems Programmer in the Networking and Telecommunications department at Rensselaer, where he continues integrating Simon with the rest of the Institute information systems. When not playing with computers, you can often find him building or renovating houses for Habitat for Humanity, as well as his own home. Reach him via USMail at RPI; VCC 319; 110 8th St; Troy, NY 12180-3590. Reach him electronically at <finkej@rpi.edu>. Find out more via <http://www.rpi.edu/~finkej>.

References

- [1] Armstrong, Eric, Steve Bobrowski, John Fazzini, Brian Linden, and Maria Pratt, *Oracle 7 Server Application Developer's Guide*, Chapter 8, pp. 1-29, Oracle Corporation, Dec 1992.
- [2] Armstrong, Eric, Steve Bobrowski, John Fazzini, Brian Linden, and Maria Pratt, *Oracle 7 Server Application Developer's Guide*, Appendix A, pp. A15-A20, Oracle Corporation, Dec 1992.
- [3] eduPerson Working Group, eduperson object class, Technical report, EDUCAUSE, <http://www.educause.edu/eduperson/>, 2001.
- [4] Finke, Jon, "Automated Userid Management," In *Proceedings of Community Workshop 1992*, Rensselaer Polytechnic Institute, pp. 3-5, Troy, NY, June 1992.
- [5] Finke, Jon, "Relational database + automated sysadmin = simon," Invited Talk, Sun Users Group, East Conference, Boston, MA, July 1993.
- [6] Finke, Jon, "Institute White Pages as a System Administration Problem," *The Tenth Systems Administration Conference (LISA 96) Proceedings*, pp. 233-240, USENIX, October 1996.
- [7] Finke, Jon, "An Improved Approach to Generating Configuration Files from a Database," *The Fourteenth Systems Administration Conference (LISA 2000)*, pp. 29-38, USENIX, December 2000.
- [8] Good, Gordon, *RFC 2849: Ldap Data Interchange Format*, June 2000.

- [9] Howe, Timothy A., *The Lightweight Directory Access Protocol: X.500 Lite*, Technical Report CITI TR 95-8, University of Michigan, July 1995.
 - [10] Hughes, Doug, "User-Centric Account Management with Heterogeneous Password Changing," *The Fourteenth Systems Administration Conference (LISA 2000)*, pp. 67-76, USENIX, December 2000.
 - [11] Microsoft, "Active directory service interfaces overview," <http://www.microsoft.com/windows2000/techinfo/howitworks/activedirectory/adsilinks.asp>.
 - [12] Microsoft, "Visual C#.net," <http://msdn.microsoft.com/vcsharp>, 2001.
 - [13] Portfolio, Tom, *PL/SQL Release 8 User's Guide and Reference*, Oracle Corporation, Part No. A58236-01, December 1997.

Stem: The System Administration Enabler

Uri Guttman – Stem Systems, Inc.

ABSTRACT

Stem is a system administration “enabler.” It is not an administrative tool, but rather a general-purpose development framework that allows an administrator to craft tools to perform a wide variety of tasks in a distributed environment easily and quickly. Many common tasks can be performed with Stem scripts involving a few lines of declarations. Current example applications include log file collating and service load balancing. Using Stem, a non-programmer can craft reliable network software in a few lines of declarations that would require hundreds or thousands of lines of code in a traditional programming language such as C.

Introduction

It is a bit difficult to begin describing something that is rather unique. Stem is not an administrative tool, nor does it support a particular kind of administrative practice. It is instead a “framework” for declarative construction of networked software for a variety of purposes. Its intent is to make network software that is as easy to create as it is to describe. In many cases, Stem allows a system administrator to create such tools without learning to program by basing them upon a few templates that describe common network applications.

Stem’s components include:

- A declarative configuration language by which one can define application components, known as “Cells.” Stem is written in Perl, and the current configuration directives use Perl syntax.
- A runtime daemon that creates and executes components (cells) based on the defined configuration.
- A set of modules that implement useful, commonly used cells.
- Additional utility modules and command tools.

While it remains a general-purpose programming framework, Stem’s primary goal is to help system and network administrators solve their problems with less work. This is the concept of “enabling” which will be a theme throughout this paper. Stem enables system administrators to easily glue existing systems together with network connections, create new networked applications with much less coding than before, and configure solutions to many common problems without any coding at all.

Stem is a general purpose system that can be used in many situations and problem spaces. By using the existing Stem modules and example configurations you can focus on your own problem space and issues and ignore many common network programming problems such as event handling and client-server and

other interprocess communication. If your problem space is large, Stem enables you to cover that entire space under one architecture, thereby simplifying your design, coding and maintenance.

To Code or Not to Code

While most administrators write at least the occasional script, many (perhaps even most) do not have the time or skills to develop full-blown network applications from scratch. Stem enables both groups of administrators equally. Non-programmers can create Stem configurations or modify the supplied examples without any coding needed to solve their specific problems, simple or complex. Administrators who know Perl can create modules for functions not provided by existing Stem modules and still take advantage of the infrastructure and network services that Stem offers, allowing them to limit their programming to just their specific task. The advantages of this approach, with respect to simplifying and speeding up application creation, are obvious. A site can even split up the work, with a developer creating new Stem modules, and an administrator creating the configuration to drive and deploy them.

Stem Networking

Gluing together disparate existing applications is a common and difficult problem when attempting to automate system administration tasks. Applications can have command line interfaces (CLI), be client/server based, or use common protocols (HTTP, SMTP, etc.). Stem enables an elegant solution to this task. Stem can be used to wrap each existing application in a module along with a new, message-based interface (essentially, an API). However, there is no need to predefine message queues or compile parameters as you do in many other message-passing or RPC type systems. Once this is done, Stem declarations invoke the module under conditions you specify.

This approach allows the task of gluing together applications to be divided into two phases: first, creating a wrapper with a message-passing interface and

second, using the new module within Stem, allowing the cells to communicate automatically with one another. The first task is performed only once, and the resulting module is reusable, and generic, while the second task allows the module to be quickly put to work on a specific task.

Related Work

Stem is a unique integration of several kinds of technology, but has its roots in several other tools that contain part, but not all, of its capabilities.

Message-passing Systems

First, Stem is a “message-passing system,” but applying that name implicitly limits it more than is appropriate. For a programmer, the term “message-passing” generally refers to parallel programming libraries such as PVM and MPI [mpi]. Stem differs from such facilities in that it allows network applications to be created at a conceptually “higher” level and handles all of the lowest level message-passing functionality transparently to the application creator. Also Stem’s messages can be sent to any cell in the current application, whether in the same hub (process), system or to a remote site. In fact, changing where a message is sent usually amounts to simply editing an address in a configuration file with no coding involved.

Another common use of the term “message-passing” is to refer to a class of commercial products commonly called Message Oriented Middleware (MOM). These products include guaranteed message delivery among their capabilities, which also typically include access to databases and transaction systems and other related services. Such products are usually targeted to the financial and business community and are rarely used by administrators and network developers. A few of the more well known MOM products include IBM’s WebSphere MQ Family and Microsoft’s MSMQ [mom].

Stem differs from MOM applications in several ways. First of all, MOM systems are designed for use by professional application programmers and usually require substantial programming expertise to use. In contrast, Stem is designed for use by administrators for administrative tasks while minimizing required programming. Secondly, traditional MOM systems are extremely large and entail considerable overhead, from both a computing and a staffing point of view. MOM systems typically need a database system to function, and some MOM vendors (e.g., IBM) recommend at least one full time staff person dedicated to running them. Stem is at the other extreme in that it is extremely lightweight. Finally, while Stem is Open Source, existing MOM applications are commercial products which are both expensive and proprietary.

Administrative Tools

Second, Stem is intended as an “administrative tool” but has almost nothing in common with existing administrative tools such as Cfengine [cfengine]. These tools are intended primarily to control the

configuration of hosts within a network. Stem is instead intended to allow flexible interoperation between tools within a network. While Cfengine provides network communications layers so that hosts can exchange information, this information is limited to facts relative to host configuration. It cannot, for example, hand off a service request from one host to another, an operation that is trivial in Stem.

The Swatch package [swatch] overlaps to some extent with one of Stem’s modules. This package monitors log file contents and searches for specified patterns set in its configuration file. The Stem::Log-Tail module performs a very similar function.

Monitoring Tools

Stem is also not a monitoring tool. It is better to say that Stem is a framework that makes it easier than ever to create applications which collect any desired system and network data. Thus, it can subsume many of the data collection capabilities of these tools. Stem could also be used to feed data to existing monitoring tools like RRDTool [rrdtool] or Cricket [cricket], enabling the administrator to extend their capabilities while continuing to take advantages of these tools’ mature visualization capabilities.

Other Facets

Stem has something in common with many other tools and approaches. Stem’s ability to function as an application wrapper has its roots in many other message-passing and “screen-scraping” systems. Its basic philosophy of creating a distributed configuration engine that responds to flexible events was first documented in Distr [distr], though this mechanism was intended solely for file distribution.

Stem Architecture

To encompass so many facets of other work, Stem has evolved a unique architecture based upon a biological metaphor of “Stem Cells.” A Cell is the fundamental building block for a network application. In a running Stem application, one or more cells exist as objects in a Stem “hub”; a hub corresponds to a single daemon process. Multiple hubs can run simultaneously, and they communicate with one another via constructs called “portals” that use TCP/IP sockets. Hubs can communicate with other hubs running on the same system or any network-accessible host, with Stem handling all of the interprocess communication.

Stem is a fully event driven system. Events can be any of the common network operations such as socket connections, I/O on character devices (terminals, sockets, pipes, etc.), and timers. Stem uses a consistent technique for the delivery of messages based on all kinds of events.

Cells

Stem cells are addressable objects which can send and receive messages. Cells are registered at creation time by name. There are three kinds of Stem Cells:

- **Class Cells** correspond to a Stem hub-wide (process) resource. These Cells are usually self-registering and generally use the class name as their identifier but more intuitive aliases may also be defined.
- **Object Cells** are application global objects. Most often, they are created and registered by the configuration which the Stem system is running, but they can also be loaded or created at runtime. They are generally long lived and last as long as the Stem hub is running.
- **Cloned Cells** are additional instances copied from an existing parent object cell. They share the parent's Cell name but are additionally given a unique target name which assigns them a unique address. Cloned cells are similar to what other systems would call sessions. They are dynamically created upon request and last only as long as needed.

The heart of Stem is the messaging subsystem and the heart of that is the registry. This is where all knowledge of how to address cells is located. Each cell is registered by its name and if it is a cloned cell, also by its target name, and messages are directed to it via these names.

Messages

Stem Messages are how Cells communicate with each other. Messages are simple data structures with two major sections, the address and the content. The address contains the Cell name that the message is directed to and which Cell sent it. The content has the message type, command and data.

Message addresses are name triplets of Hub/Cell/Target. The Cell name is required and the Hub and Target are optional. These triplets form globally unique addresses with the overall Stem system.

The Message address section has multiple address fields. The two primary fields correspond to the common email headers and are called 'to' and 'from'. The 'to' address designates which Cell will get this message, and the 'from' address says which Cell sent this message.

The Message content has information about this message and any data being sent to the destination Cell. The primary attribute is 'type' which can be set to any string, but common types are data, cmd (command), response and status. Stem modules and Cells can create any Message types they want. The other major attribute of the content is the data, which holds a reference to the message data.

Modules

The various Cells classes are implemented as Perl modules, and many useful Cell types are included with the Stem package. These are among the most important:

- **Stem::SockMsg:** Cells that connect to/accepts connections from sockets. These cells function as a socket-to-Stem message gateway.

- **Stem::Switch:** Multiplexes Stem messages to multiple destinations according to maps which can be dynamically modified.
- **Stem::Portal:** Manages connections between Stem hubs, facilitating message transmission across the network (including authentication and security functions).
- **Stem::TtyMsg:** Provides a TTY interface to a Stem hub.
- **Stem::Proc:** Creates Cells that fork external processes and manages them.
- **Stem::Log:** Writes and manages Stem logs (which may be associated with external files).
- **Stem::Log::Tail:** Monitors active external files (typically log files), sending newly acquired data into the Stem logging subsystems on either a periodic basis or on demand.
- **Stem::Cron:** Creates and manages scheduled message submissions. Such messages can be sent anywhere in a Stem network and can trigger any Stem operation. If the message is addressed to a Stem::Switch Cell, it can then be sent to multiple destinations and trigger events across the network from a centralized schedule.

In addition, Stem includes other utility modules which provide services to active Cells. These services include asynchronous I/O, cloning of cells, flow control or local and remote method calls and logical pipes.

Configurations

Stem differs from all other networking toolkits by being architected around configuration rather than software. A configuration file instructs the Stem engine which Stem cells to construct and register. When you invoke Stem, it interprets this configuration and creates cells as needed without any need to compile networking code.

Configuration files are structured using Perl objects. Each desired cell is specified as a Perl object with a list of attributes. Several examples are discussed in the next section.

Example Application: An inetd-like Server

We will consider a few versions of a small Stem application, chosen for its use of typical Stem components as well as in response to the space limitations imposed on this paper. None of them require any programming on the part of the system administrator.

A Simple First Version

This version of the application serves as a good starting point for understanding Stem. It uses only existing Stem modules to create an application which can execute a process on a local or remote system. As such, the only task required to create the application is to set up a file containing the Stem configuration. The simplest version is given in Listing 1.

This configuration uses three cells:

- A Stem::TtyMsg Cell. This Cell is used to enable commands for the hub to be typed in at the keyboard. In this configuration the default attributes are used.
- A Stem::Proc Cell named “mon”. This section of the configuration file will create a Cell that starts a process on demand. Here we specify a list of arguments for the Cell: the path to the command to be run, and some generic Cell attributes. The latter specify that the cell is to be cloned and that it will only send the data from its process when it exits.

```
# uptime.stem
[
    class => 'Stem::TtyMsg',
    args => [],
],
[
    class => 'Stem::Proc',
    name => 'mon',
    args => [
        path      => '/usr/bin/uptime',
        cell_attr => [
            'data_addr' => 'A',
            'send_data_on_close' => 1,
        ],
    ],
],
[
    class => 'Stem::SockMsg',
    name => 'A',
    args => [
        port      => 6666,
        server    => 1,
        cell_attr => [
            'data_addr' => 'A',
        ],
    ],
].
```

Listing 1: Simplest configuration file.

- A Stem::SockMsg Cell named “A”. This Cell will listen on a socket (the port address is specified by the port attribute). When a connection request is accepted, it will create a logical pipe to the ‘mon’ cell as specified in the ‘pipe_addr’ attribute.

In this example when a socket connection is made, the logical pipe to ‘mon’ is created, which causes the ‘mon’ cell to clone and fork the uptime program. Its output is collected and then sent back to the ‘A’ send and then on to the socket. Stem will take care of creating and sending all of those messages automatically.

Running this configuration requires the following commands:

```
$ xterm -T Stem -n Stem \
-e run_stem uptime
$ xterm -T Monitor -n Monitor \
-e telnet localhost 6666
```

We use two xterm commands to make Stem’s operations visible. The first command starts the Stem hub daemon process and attaches the Stem::TtyMsg Cell to it so commands can be entered. The second command attaches a second window to port 6666, the Stem::SockMsg Cell, using a telnet command. Figure 1 illustrates the resulting windows.

In the “Stem” window, we send the `cell_trigger` command to the mon Cell. This causes the Stem::Proc Cell to execute its command. Note that the output appears in the “Monitor” window attached to port 6666 each time the cell is triggered.

A Piped Version

The problem with the example above is that you must enter a `cell_trigger` command to make the process execute and only one telnet session can be used. This new version (see Listing 2) will make the process execute when the telnet connects to the socket. Note it uses the `pipe_addr` attribute which will create a logical pipe between the cell ‘A’ and the ‘mon’ cell. Actually the ‘mon’ cell will be cloned when a pipe to it is created and that cloned cell will run the process.

To run this example, name the configuration file up3.stem and run this command:

```
$ xterm -T Stem -n Stem \
-e run_stem uptime2
```

Then, in another window run the telnet command:

```
$ telnet localhost 6666
```

Each time you run telnet you will invoke the `uptime` command and see its output from telnet which will then exit.

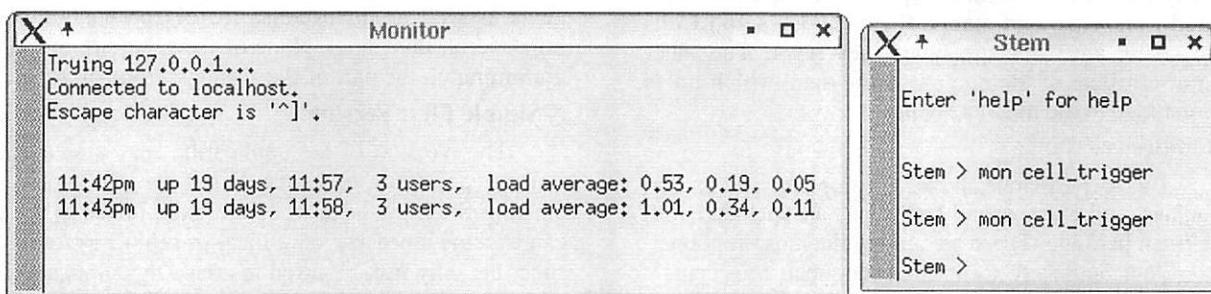


Figure 1: A simple Stem application.

A Multi-Hub Version

The previous two Stem applications were extremely simple, but their potential functionality is very powerful. They can be extended in several ways:

- The process can be executed on a different host than the triggering host.
- The process can be executed on multiple remote systems.
- The output can be sent to more than one destination: multiple sockets on different systems, a log file, another Stem cell for further processing, and so on.
- A different command or program can be run. The uptime command merely serves as a proof-of-concept here. Any command that is needed could be executed.

```
#uptime2.stem
[
    class => 'Stem::TtyMsg',
    args  => [],
],
[
    class => 'Stem::Proc',
    name  => 'mon',
    args  => [
        path      => '/usr/bin/uptime',
        cell_attr => [
            'cloneable' => 1,
            'send_data_on_close' => 1,
        ],
    ],
],
[
    class => 'Stem::SockMsg',
    name  => 'A',
    args  => [
        port      => 6666,
        server    => 1,
        cell_attr => [
            pipe_addr => 'mon',
        ],
    ],
]
```

Listing 2: Improved with concurrent sockets.

- More than one command can be supported by defining multiple Stem::Proc cells. In this way, the application can function in a similar way to inetd in that it can start any one of a number of preconfigured servers upon demand.
- The process can be triggered other ways than by manually entering a command or by connecting to a socket: by a different Stem cell, according to a schedule (using Stem::Cron), etc. The triggering can come from the server hub, the client hub, or elsewhere in the Stem system.

Listings 3 and 4 illustrate a multi-hub version of this application. Notice how easy it is to split the simple version into an implementation which can be run across the network.

To run this application, start processes like these (as before):

```
$ xterm -T Server -n Server \
-e run_stem uptime_server
$ xterm -T Client -n Client \
-e run_stem uptime_client
```

Then, in another window run this command:

```
$ telnet localhost 6666
```

This will behave the same as the uptime2 example but it is split over two hubs. Notice that other than adding the Hub and Portal cells, the only change to the configuration was adding a hub name to the 'pipe_addr' attribute in the uptime_client configuration. This illustrates how easy it is to distribute applications written in Stem across a network. Sending messages to local or remote cells is done the same way – typically only the address will need to be changed.

This example application and its variations provide some insight into Stem's flexibility and capabilities. The Stem distribution comes with several other example applications and a cookbook that shows you how to create your own cells.

```
#uptime_server.stem
# Name the hub so the client can
# refer to it.
[
    class => 'Stem::Hub',
    name  => 'uptime_server',
    args  => [],
],
[
    class => 'Stem::TtyMsg',
    args  => [],
#
# Set the portal to be a server:
# listen for portal connections from
# any host the port defaults to 10000
# but can be set here
[
    class => 'Stem::Portal',
    name  => 'listener',
    args  => [
        'server'  => 1,
        'host'    => '',
    ],
],
[
    class => 'Stem::Proc',
    name  => 'mon',
    args  => [
        path      => '/usr/bin/uptime',
        cell_attr => [
            'cloneable' => 1,
            'send_data_on_close' => 1,
        ],
    ],
]
```

Listing 3: Multi-hub server.

Critique and Analysis

The major infrastructure work of creating Stem has been completed, and the package is working well where it has been deployed. The design has proven to be as flexible as was intended, and Stem applications have been created by administrators unfamiliar with the package after just a few hours.

Stem's planned extensibility has also been verified in that administrators have successfully written additional Stem modules in Perl and integrated them with those that the package provides.

Stem's highly modular design has been proven to be instrumental in extending it. New modules can easily be created and integrated. Internal services have been developed and quickly used by other modules and cells. Its simple message-passing API allows almost any external application, service or protocol to interact with any other.

```
#uptime_client.stem
# Name the hub so server can refer to it.
[
  class => 'Stem::Hub',
  name  => 'uptime_client',
  args   => [],
],
[
  class  => 'Stem::TtyMsg',
  args   => [],
].
# Create a client portal.
# this will connect to the portal
# in the 'monitoring' hub the default
# host is 'localhost' but it can be
# set here the port defaults to
# 10000 but can be set here
[
  class => 'Stem::Portal',
  name  => 'server',
  args   => [],
],
[
  class  => 'Stem::SockMsg',
  name  => 'A',
  args   => [
    port      => 6666,
    server    => 1,
    cell_attr => [
      cloneable => 1,
      pipe_addr => 'uptime_server:mon',
    ],
  ],
].

```

Listing 4: Multi-hub client.

Stem's configuration file format currently takes the form of Perl data structures. This has performance implications and security issues as well as presenting a somewhat eccentric interface to non-Perl literate

system administrators. In the future, we plan to support multiple configurations formats including XML.

Similarly the format of Stem's message (when serialized over a pipe) also needs to support other formats. But as with the configuration formats, it is just a matter of having modules that can convert the internal message structure to/from an external format. This will allow other systems to be more easily integrate as they can then send/receive Stem messages.

Stem's security support currently is weak. We have demonstrated to ourselves that we can use ssh for message-passing between Stem processes but the design was not good enough for production. We have plans to redesign it to be integrated with Stem's socket module so that any IPC (not just message-passing) can use it. Also the design would allow a choice of secure transport (ssh, SSL etc.) by using the same modular plug-in design as mentioned above.

Future Work

Stem is extremely modular in design and can be extended easily in many directions. Here is a short list of some items that are in our development queue now:

- State Machine: A text based state machine that will take input from multiple sources: socket, processes and messages. It will have many features including state callbacks, input and output buffers, regular expression matching, and the like.
- Flow Control: A module that will allow a Stem Cell to control the logic flow of method calls, regardless of whether they are local or remote. It manages a combination of synchronous (local) and asynchronous (remote via messages) object method calls in a simple mini-language that will have the common flow control operations such as IF/ELSE, WHILE, etc. This greatly simplifies the task of coordinating distributed operations upon an object (a Stem Cell) such as accessing a database or using subprocesses and remote protocols.
- Network Protocols: When the State Machine is finished, it will be used in various protocol modules which will enable Stem to communicate with programs which use the popular protocols such as HTTP, FTP, SMTP, etc.
- GUI-based Configuration Tool: A longer term goal is to integrate Stem with a GUI toolkit such as Tk or Qt in order to develop a tool for creating and visualizing Stem configurations. Doing so will also make it possible to create a wide range of GUI front ends for Stem based applications.

Availability

Stem is Open Source software, it is licensed under the GPL and is available without charge from Stem Systems. Our website is <http://www.stemsystems.com>.

Acknowledgments

I wish to thank Aeleen Frisch, Alva Couch, and Will Partain for their help with this paper.

Author biography

Uri Guttman graduated from MIT in 1983 with a B.S. CSE. He has been developing software for 25 years. Stem is the result of his extensive experience in systems architecture, networking, communications, API design and Perl. He is currently president of Stem Systems and can be reached at uri@stemsystems.com .

References

- [cfengine] Burgess, Mark, "Cfengine: A Site Configuration Engine," *USENIX Computing Systems*, USENIX, 1995.
- [cricket] Allen, Jeff R., "Driving by the Rear-View Mirror: Managing a Network with Cricket," *Proceedings First Conference on Network Administration*, USENIX 1999.
- [distr] Couch, Alva L., "Chaos Out of Order: A File Distribution Facility For 'Intentionally Heterogeneous' Networks," *Proceedings LISA 1997*, USENIX, 1997.
- [mom] <http://www-3.ibm.com/software/ts/mqseries/> and <http://www.microsoft.com/msmq/default.htm>.
- [mpi] The Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard" and "MPI-2: Extensions to the Message-Passing Interface," <http://www mpi-forum.org>.
- [rrdtool] Oetiker, Tobias, "RRDTool," <http://people.ee.ethz.ch/~oetiker/webtools/rrdtool/manual/index.html>.
- [swatch] Hansen, Stephen E., and E. Todd Atkins, "Centralized System Monitoring With Swatch," *Proceedings LISA 1993*, USENIX, 1993.

Pan: A High-Level Configuration Language

Lionel Cons and Piotr Poznański – CERN, European Organization for Nuclear Research

ABSTRACT

The computational requirements for the new Large Hadron Collider are enormous: 5-8 PetaBytes of data generated annually with analysis requiring 10 more PetaBytes of disk storage and the equivalent of 200,000 of today's fastest PC processors. This will be a very large and complex computing system, with about two thirds of the computing capacity installed in "regional computing centres" across Europe, America, and Asia.

Implemented as a global computational grid, the goal of integrating the large geographically distributed computing fabrics presents challenges in many areas, including: distributed scientific applications; computational grid middleware; automated computer system management; high performance networking; object database management; security; global grid operations.

This paper describes our approach to one of these challenges: the configuration management of a large number of machines, be they nodes in large clusters or desktops in large organizations.

Introduction

The European Organization for Nuclear Research (CERN) is building the Large Hadron Collider (LHC), the world's most powerful particle accelerator. From the LHC Computing Grid Project home page (<http://cern.ch/LHCgrid>):

The computational requirements of the experiments that will use the LHC are enormous: 5-8 PetaBytes of data will be generated each year, the analysis of which will require some 10 PetaBytes of disk storage and the equivalent of 200,000 of today's fastest PC processors. Even allowing for the continuing increase in storage densities and processor performance this will be a very large and complex computing system, and about two thirds of the computing capacity will be installed in "regional computing centres" spread across Europe, America and Asia.

The computing facility for LHC will thus be implemented as a global computational grid [10], with the goal of integrating large geographically distributed computing fabrics into a virtual computing environment. There are challenging problems to be tackled in many areas, including: distributed scientific applications; computational grid middleware; automated computer system management; high performance networking; object database management; security; global grid operations.

This paper describes our approach to one of these challenges: the configuration management of a large number of machines, be they nodes in large clusters or desktops in large organizations.

Large Scale System Administration

Many solutions for managing a few machines do not scale well. When dealing with thousands of machines, some problems start to overwhelm.

Automation

It is fine to reinstall your home PC by booting from an installation diskette and typing a few commands but you certainly do not want to do it on a large cluster. Similarly, it is acceptable to log into one machine and purge /tmp by hand but this could also be automated using a tool like Red Hat's tmpwatch.

Technical solutions exist in many domains (remote power control, console concentration, network booting, unattended system installation, package management, etc.) so manual interventions can and should be limited to the absolute minimum. What remains to be done by hand is to configure the programs that will automate these otherwise manual tasks.

This approach has an interesting side effect. System configurations are known to rot with time because ad hoc system interventions tend to accumulate small mistakes until the system malfunctions. An unattended but complete reinstallation from scratch (not a restore from backup) is the most cost effective way to get rid of the problem. All you need is to make sure that the configuration of the installer is kept up to date, which is not difficult to do.

Abstraction

Once we have reached the full automation of the installation process, we can define the configuration of a machine as the sum of all the configurations of all the programs used either during the installation or afterwards. This will contain everything from disk partitioning information to system or software configuration (networking, user accounts, X server, etc.).

You should not include all the things that *can* be configured but rather the ones that *will* be configured. For instance, if you do not need to change the /usr/share/magic file used by the file command, consider it as a (static) data file that comes with the file package

itself and therefore outside of the machine configuration abstraction.

We then want to reason about these machine configurations and, for instance, express the fact that two machines have the same disk model or the fact than one thousand machines belong to the same batch cluster. It is tedious to use the native configuration files, e.g., `/etc/services` to describe the known network services or `/etc/crontab` for cron's configuration, because of the duplication of information and the variety of convoluted formats. We really need an abstraction of this information that is easy to use.

The way the configuration information is abstracted and represented is very important. This is in line with Eric S. Raymond's advice in *The Cathedral and the Bazaar* [16]: "Smart data structures and dumb code works a lot better than the other way around."

It is quite expensive to come to a good abstraction but it really pays off when managing many machines. It is a virtuous circle: the more data you put in the abstraction, the more useful it becomes.

Single Database for Multiple Tools

In theory, a unique system administration tool is better than a set of unrelated and often overlapping tools such as AutoRPM, cfengine, RDist, LCFG, etc. In practice, such a mythical beast does not exist and system administrators use the tools' combination which is adapted to their needs, often with a pinch of home made scripts with "glue languages" such as Perl or TCL.

It is good to combine the strengths of these tools, but the variety of their configuration formats is a big disadvantage. Information is duplicated and often cumbersome to maintain. Until recently, the machines in our computer centre were drawing on information from more than twenty different sources, from flat files to real databases. Mistakes when handling these files (e.g., adding a machine and forgetting to update one file) were a common source of problems.

A good approach is to use a single source of information (a central configuration database) and simple programs that can transform this information into the format understood by the tools used.

Change Management

In the first eight months of its life, Red Hat Linux 7.2 had 311 updated RPMs. This is more than one updated RPM per day on average. Just looking at security, Red Hat issued 59 security advisories for the same system during the same period, almost two per week on average. In large computing centres, changes will occur frequently so you must manage them adequately.

Every component (be it hardware or software) has a non-zero failure probability so, statistically, large computer centres have a high probability of having, at any point in time, one or more components not

working properly. This is especially true when using cheap commodity hardware. Some machines will always be down or somehow unreachable, so configuration changes have to be deployed asynchronously.

Moreover, some critical processes cannot be interrupted and intrusive system management tasks (such as changing the kernel in use) have to be deferred until the system is ready to accept the changes.

The consequence is that you should not try to configure a machine directly but rather change its configuration (stored outside of the machine) and let the machine bring itself into line whenever it can. This can be called convergent or asymptotic configuration (see for instance [18]): the machines independently try to come closer to their "desired state."

Validation

We should keep in mind this slight modification of one of Murphy's laws: anything that can go wrong will go wrong more spectacularly with central system administration.

Having a central database holding machine configurations and letting thousands of programs on remote machines use it is very powerful but mistakes can have disastrous consequences. It is of paramount importance to control the changes and detect mistakes before it is too late. Advanced means of validating the stored information must be in place.

The good news is that the abstraction mentioned above really helps. Once you can reason about machines and their configuration parameters, it is easy to express constraints such as "for all the machines, the filesystems mounted through NFS must be exported by the corresponding server." The example given in Appendix C describes exactly this constraint in our Pan language.

Validation should not only be seen as a way to prevent mistakes, it can also be used to make sure that things are really the way you want them to be. Constraints can be used, for example, to ensure that all machines have enough swap space, or that they are running the correct version of some software.

Our Solution

Overview

The Fabric Management Work Package (WP4 [21]) of the European Union DataGrid Project (EDG [8]) seeks to control large computing fabrics through the central management of their "desired state" via a central configuration database (one per administrative domain). This information will then be used in different ways.

For the initial system installation (we currently use Red Hat Linux 7.2), it will be used to create the various files needed to fully automate this process, for instance DHCP entries and Kickstart files.

For the system maintenance (we currently use LCFG), the configuration information will be directly used by a number of modular “component” scripts which are responsible for different subsystems, such as “mail configuration” or “web server configuration.” The components are notified when their configuration changes and are responsible for translating the abstract configuration into the appropriate configuration files, and reconfiguring any associated daemons.

Machines will be self healing thanks to sensors reporting information to a monitoring database and actuators using this information to trigger recovery actions such as restarting a daemon or, in extreme cases, triggering a full reinstallation of the machine. Through the inclusion of hardware information in the configuration database, we can also detect problems such as a dead CPU or stolen memory.

Configuration Database

The configuration database [4] stores two forms of configuration information. One is called the High Level Description [5] and is expressed in the Pan language. The other is the Low Level Description [11] and is expressed in XML. Both are explained below.

The system administrators can edit the High Level Description, either directly or through some scripting layer. The Low Level Description (one XML file per machine) is always generated using the Pan compiler.

The XML machine configuration is cached on the machine (to support disconnected operations) and access is provided through a high-level library [15] that hides the details such as the XML schema used.

The database itself includes a scalable distribution mechanism for the XML files based on HTTP, and the possibility of adding any number of backends (such as LDAP or SQL) to support various query patterns on the information stored. It should scale to millions of configuration parameters.

Low Level Description

Mapping a configuration abstraction to a tree structure is quite easy. This is the natural format for most “organized information,” from files in a filesystem to the Windows registry or LDAP. We call this the Low Level Description (LLD).

Simple values (like strings or numbers) form the leaves of this tree and are called *properties*. Internal nodes of the tree are called *resources* and are used to group *elements*¹ into *lists* (accessed by index) or named lists (aka *nlists*, accessed by name). Nlists can conveniently be used to represent tables or records². Every element has a unique *path* which identifies its position in the tree.

We chose XML to represent this tree in a file because it maps well to the hierarchical structure of

¹The term *element* refers to either a property or a resource.
²I.e., similar to Pascal’s record or C’s struct.

the information and it is easy to parse and to validate (with XML Schema). Here is a small example representing some hardware information (a larger example can be found in Appendix A):

```
<?xml version="1.0" encoding="utf-8"?>
<nlist name="profile">
  <nlist name="hardware">
    <nlist name="memory">
      <long name="size">512</long>
    </nlist>
    <nlist name="cpus">
      <nlist>
        <string name="vendor">
          Intel
        </string>
        <string name="model">
          Pentium III (Coppermine)
        </string>
        <double name="speed">
          853.22</double>
      </nlist>
    </list>
  </nlist>
</nlist>
</nlist>
```

Starting with the toplevel XML element (named “profile”), you can see on the fifth line the property describing the memory size: its path is /hardware/memory/size and its value is the long integer 512. Similarly, /hardware/cpus is the resource representing the list of CPUs. The path /hardware/cpus/0³ identifies the first (and only) CPU which is represented using a nlist that holds a kind of record or structure describing the CPU. The model of the CPU is the string at path /hardware/cpus/0/model and its value is “Pentium III (Coppermine).”

The way this information appears in the XML file is dependent on the programs using it. For instance, if you have only a few X server configurations but a large variety of resolution settings, you could have something like:

```
<nlist name="profile">
  <nlist name="system">
    <nlist name="x">
      <string name="XF86Config"
        type="fetch">
        http://config.cern.ch/XF86Config-ATI64-19
      </string>
      <list name="modes">
        <string>1280x1024</string>
        <string>1024x768</string>
      </list>
    </nlist>
  </nlist>
</nlist>
```

The special *fetch* type is known by our system and the programs accessing the configuration information through our API will simply see the contents of the file at the given URL as a string. A program

³In paths, numbers are used to identify list items, the first one having the index 0, like in C.

responsible for managing the X configuration would simply have to start with this base file, substitute in the desired modes and write the result to /etc/X11/XF86Config.

High Level Description

Although the previous XML representation is sufficient for the programs running on the target machines, we need a High Level Description (HLD) to reason about groups of machines and share common information.

Existing tools (such as m4) only cover some of our requirements so we decided to design our own language to represent the HLD and we wrote the accompanying compiler transforming this HLD into LLD (i.e., XML).

We believe (in line with Paul Anderson's *A Declarative Approach to the Specification of Large-Scale System Configurations* [2]) that a declarative approach⁴ to configuration specification is better suited than a procedural one⁵. Pan has been designed to stay as declarative as possible while allowing some form of procedural code, which is required to take full advantage of the power of validation.

The following is a quick overview of the salient features of the Pan language. The complete language specification is available in another document [5].

⁴I.e., describe how things should look like in the end.

⁵I.e., describe the sequence of actions to be performed.

```
# definition for the disk IBM DTLA-307030
structure template disk_ibm_dtl_a_307030;
"type"      = "disk";
"vendor"    = "IBM";
"model"     = "DTLA-307030";
"size"      = 29314; # MB

# definition for the hardware Elonex 800x2/512
structure template pc_elonex_800x2_512;
"vendor"    = "Elonex";
"model"     = "800x2/512";
"cpus"       = list(create("cpu_intel_p3_800"), create("cpu_intel_p3_800"));
"memory/size" = 512; # MB
"devices/hda" = create("disk_ibm_dtl_a_307030");

# definition for the Venus cluster
template cluster_venus;
"/hardware" = create("pc_elonex_800x2_512");
# and any other hardware or system information shared by all the
# members of the Venus cluster

# first machine
object template venus001;
include cluster_venus;
"/hardware/serial" = "CH01112041";

# second machine
object template venus002;
include cluster_venus;
"/hardware/serial" = "CH01117031";
# the first disk has been replaced
"/hardware/devices/hda" = create("disk_quantum_fireballp_as20_5");
```

Listing 1: Two cluster nodes which share most of their configuration.

Overview

Pan mainly consists of *assignments*, each of which sets some value in a given part of the LLD identified by its path. The following code can be used to generate the LLD shown earlier. The left hand side of the assignment is the path and the right hand side is the value. *nlist* is a builtin function that will return the nlist made from its arguments.

```
"/hardware/memory/size" = 512;
"/hardware/cpus/0" = nlist(
  "vendor", "Intel",
  "model", "Pentium III (Coppermine)",
  "speed", 853.220,
);
```

Pan also features other statements like *include* (very similar to cpp's *#include* directive) or *delete* that can delete a part of the LLD.

The grouping of statements into *templates* allows the sharing of common information and provides a simple inheritance mechanism. A *structure template* is used to represent a subtree of information (for instance a given disk) while an *object template* represents a real world object (the compiler will generate a separate LLD for every object template encountered).

Listing 1 shows a partial example of two cluster nodes that share most of their configuration information.

Types

Pan contains a very flexible typing mechanism. It has several builtin types (such as boolean, string, long,

...) and allows compound types to be built on top of these. Once the type of a configuration element is known, the compiler makes sure that only values of the right type are assigned to it. By explicitly specifying the type of the root element (i.e., the top of the configuration tree), one can completely define the schema of the information that is found in the LLD. The type enforcement done by the compiler guarantees that only LLDs conforming to the schema will be generated. This enforcement is illustrated in Listing 2.

Starting with the root of the LLD, the compiler will make sure that the data corresponds to the declared type. Extra and missing fields in structures trigger a compilation error. The code in Listing 2 ensures that /hardware/memory/size is always present and contains a positive long integer.

Validation

To have even greater control on the information generated by the compiler, one can attach arbitrary

```
# structure representing the (physical) memory
define type memory_t = {
    "size" : long(0...)          # a long which is greater than 0
};

# structure representing the complete hardware
define type hardware_t = {
    "vendor"   : string
    "model"    : string
    "serial"   ? string         # this field is optional
    "memory"   : memory_t
    "cpus"     : cpu_t[1..8]    # a list of between 1 and 8 cpu_t
    "devices"  : device_t{}    # a (maybe empty) table of device_t
};

# structure representing the root of the configuration tree
define type root_t = {
    "hardware" : hardware_t
    "system"   : system_t
    "software" : software_t
};

# the root of the configuration tree (i.e., /) must be of type root_t
type "/" = root_t;
```

Listing 2: Type enforcement.

```
# IPv4 address in dotted number notation
define type ipv4 = string with {
    result = matches(self, '^(\d+)\.(\d+)\.(\d+)\.(\d+)$');
    if (length(result) == 0)
        return("bad string");
    i = 1;
    while (i <= 4) {
        x = to_long(result[i]);
        if (x > 255)
            return("chunk " + to_string(i) + " too big: " + result[i]);
        i = i + 1;
    };
    return(true);
};

# make sure that we have at least 256MB of RAM per processor
valid "/hardware/memory/size" = self >= 256 * length(value("/hardware/cpus"));
```

Listing 3: Validation code.

validation code either to a type or to a configuration path; see Listing 3.

Data Manipulation Language

The validation code is represented in a simple yet powerful data manipulation language which is a subset of Pan and syntactically similar to C or Perl. Rather than embedding another language such as Perl or Python for this task, we decided to design our own. This was necessary to maintain control over type checking and to encourage users to use the declarative parts of Pan. Built-in functions such as pattern matching and substitution are available and user-defined functions are supported.

Although we prefer the declarative approach to the procedural approach, this data manipulation language is very convenient to perform complex operations. Listing 4 illustrates the use of Pan to introduce an element into a given list position.

Miscellaneous

The Pan compiler keeps track of derivation information which precisely links configuration information appearing in the LLD to the originating HLD statements. When HLD templates are modified, this derivation information is used to determine which LLDs must be recreated, thus minimizing the work carried out by the compiler. The information is also used to determine which HLD templates are responsible for the final value of a given configuration parameter.

Comparison With Other Tools

Pan (and its associated compiler) cannot be considered as a system administration tool by itself: it is only a language to express configuration information. As far as we know, there exists no similar tool to compare directly with. What follows is a comparison with the way different tools or projects manipulate system configuration information.

Arusha Project

The Arusha Project (ARK, <http://ark.sourceforge.net>) provides a framework for collaborative system administration [13]. It provides a simple, XML-based language that can be used to describe almost everything, from package management to documentation or system configuration. Unfortunately, this language lacks strong type checking and validation. It also mixes code and data (e.g., some Perl or Python code can be embedded inside XML, close to the data) which is something that we do not want.

Cfengine

Cfengine (<http://www.cfengine.org>) is an autonomous agent [3] with a high level declarative language to manage large computer networks. It has no real types and a limited support for lists. Its configuration is not a real abstraction of the machine configuration but rather some instructions for its different modules such as

```
# insert a string after another one in a list of strings
# (or at the end if not found)
define function insert_after = {
    if (argc != 3 || !is_string(argv[0]) || !is_string(argv[1]) ||
        !is_list(argv[2]))
        error("usage: insert_after(string, string, list)");
    idx = index(argv[1], argv[2]);
    if (idx < 0) {
        # not found, we insert at the end
        splice(argv[2], length(argv[2]), 0, list(argv[0]));
    } else {
        # found, we insert just after
        splice(argv[2], idx+1, 0, list(argv[0]));
    }
    return(argv[2]);
};

# here is how to use it to insert "apache" after "dns"
"/boot/services" = list("dns", "dhcp", "mail", "postgres");
"/boot/services" = insert_after("apache", "dns", value("/boot/services"));
```

Listing 4: Introducing an element into a given list position.

network interface configuration, symbolic links management, checks for permissions and ownership of files, etc. For instance, the modification of system files like /etc/inetd.conf is often done with instructions such as AppendIfNoSuchLine or CommentLinesMatching. It has no support for validation.

DMTF

The Distributed Management Task Force (DMTF, <http://www.dmtf.org>) is an organization developing “management standards.” The standards closest to Pan are part of the Common Information Model (CIM, http://www.dmtf.org/standards/standard_cim.php). Their approach is complex and mixes configuration management and system monitoring. Although their standards could not be used directly inside our work, we have tried to stay close and to reuse some parts of their data schemas.

LCFG

LCFG [1] (<http://www.lcfg.org>) is a system for automatically installing and managing the configuration of large numbers of Unix systems. It does use some abstraction to describe the machine configuration but the language used does not really have types. All the parameters are basically strings similar to X resources and compound types (such as lists or tables) are built on top of these with some ad-hoc name mangling. The inheritance is achieved by using cpp and include files. The combination of cpp macros and embedded Perl code hinder the clarity of this otherwise mainly declarative language. On the other hand, it has some advanced features (like constraint based list ordering) that will probably be added to Pan in the future.

Although LCFG and EDG are separate projects, the development teams share ideas and some compatibility exists. For instance, the Pan compiler can produce some XML files that can be understood by the LCFG components.

Status and Availability

After a first Perl prototype last year, the new compiler (built using C++, STL, Lex and Yacc) is almost complete (at the time of this writing) and will be delivered to the EDG in September.

The software will be available under the open source EDG Software License⁶ from the EDG WP4 Configuration Task web site at <http://cern.ch/hep-proj-grid-fabric-config>.

At the time of this writing, the Pan language has been successfully used to describe a large fraction of the configuration of the Linux machines used inside the EDG project. Work is in progress to extend this to other machines in our computer centre at CERN.

Acknowledgements

We would like to thank the European Union for their support of the EDG project and our colleagues from the WP4 for their contributions and very fruitful discussions on the topics of system administration and configuration.

Authors Biographies

Lionel Cons earned an “Ingénieur de l’École Polytechnique” (Paris) diploma in 1988 and the ENSIMAG’s engineer’s diploma two years later. He then joined CERN where he worked as a C software developer and then as a UNIX system engineer in the Information Technology division. He presently works on system security and is the leader of the WP4 configuration task of the EDG project.

Piotr Poznański earned an MSc degree in Computer Science from the University of Mining and Metallurgy (Cracow, Poland). He joined CERN in 2000 and currently works in the EDG project as a software engineer.

References

- [1] Anderson, Paul, “Towards a High-Level Machine Configuration System,” *LISA Conference Proceedings*, 1994.
- [2] Anderson, Paul, *A Declarative Approach to the Specification of Large-Scale System Configurations*, <http://www.dcs.ed.ac.uk/home/paul/publications/conflang.pdf>, 2001.
- [3] Burgess, Mark, “Computer Immunology,” *LISA Conference Proceedings*, 1998.
- [4] Cons, Lionel and Piotr Poznański, *Configuration Database Global Design*, <http://cern.ch/hep-proj-grid-fabric-config>, 2002.
- [5] Cons, Lionel and Piotr Poznański, *High Level Configuration Description Language Specification*, <http://cern.ch/hep-proj-grid-fabric-config>, 2002.
- [6] da Silva, Fabio Q. B., Juliana Silva da Cunha, Danielle M. Franklin, Luciana S. Varejao, and Rosalie Belian, “A Configuration Distribution System for Heterogeneous Networks,” *LISA Conference Proceedings*, 1998.
- [7] da Silveira, Gledson Elias and Fabio Q. B. da Silva, “A Configuration Distribution System for Heterogeneous Networks,” *LISA Conference Proceedings*, 1998.
- [8] European Union DataGrid Project (EDG), <http://www.eu-datagrid.org>.
- [9] Evard, Rémy, “An Analysis of UNIX System Configuration,” *LISA Conference Proceedings*, 1997.
- [10] Foster, Ian and Carl Kesselman, *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, http://www.mkp.com/books_catalog/catalog.asp?ISBN=1-55860-475-8, 1998.
- [11] George, Michael, *Node Profile Specification*, <http://cern.ch/hep-proj-grid-fabric-config>, 2002.
- [12] Harlander, Dr. Magnus, “Central System Administration in a Heterogeneous Unix Environment: GeNUAdmin,” *LISA Conference Proceedings*, 1994.
- [13] Holgate, Matt and Will Partain, “The Arusha Project: A Framework for Collaborative Unix System Administration,” *LISA Conference Proceedings*, 2001.
- [14] Large Scale System Configuration Workshop, <http://www.dcs.ed.ac.uk/home/dcspaul/wshop>, 2001.
- [15] Poznański, Piotr, *Node View Access API Specification*, <http://cern.ch/hep-proj-grid-fabric-config>, 2002.
- [16] Raymond, Eric S., *The Cathedral and the Bazaar*, O'Reilly, <http://www.oreilly.com/catalog/cb>, 1999.
- [17] Rouillard, John P. and Richard B. Martin, “Config: A Mechanism for Installing and Tracking System Configurations,” *LISA Conference Proceedings*, 1994.
- [18] Sventek, Joe, *Configuration, Monitoring and Management of Huge-scale Applications with a Varying Number of Application Components*, <http://www.dcs.ed.ac.uk/home/dcspaul/wshop/HugeScale.pdf>, 2001.
- [19] Traugott, Steve and Joel Huddleston, “Bootstrapping an Infrastructure,” *LISA Conference Proceedings*, 1998.
- [20] van der Hoek, André, Dennis Heimbigner, and Alexander L. Wolf, *Software Architecture, Configuration Management, and Configurable Distributed Systems: A Ménage à Trois*, <http://citeseer.nj.nec.com/hoek98software.html>, 1998.
- [21] WP4, *EDG Fabric Management Work Package*, <http://cern.ch/hep-proj-grid-fabric>.
- [22] WP4C, *EDG WP4 Configuration Task*, <http://cern.ch/hep-proj-grid-fabric-config>.

⁶<http://www.eu-datagrid.org/license.html>.

Appendix A: Partial LLD Example

This is an oversimplified example; more complete examples can be found on our web site [22].

```

<?xml version="1.0" encoding="utf-8"?>
<nlist name="profile" type="record">
  <nlist name="hardware" type="record">
    <string name="vendor">Elonex</string>
    <string name="model">850/256</string>
    <list name="cpus">
      <nlist type="record">
        <string name="vendor">Intel</string>
        <string name="model">Pentium III (Coppermine)</string>
        <double name="speed">853.22</double>
      </nlist>
    </list>
    <string name="serial">CH01112041</string>
    <nlist name="memory" type="record">
      <long name="size">256</long>
    </nlist>
    <nlist name="devices" type="table">
      <nlist name="hda" type="record">
        <string name="vendor">QUANTUM</string>
        <string name="model">FIREBALL P AS20.5</string>
        <string name="type">disk</string>
        <long name="size">19596</long>
      </nlist>
      <nlist name="hdc" type="record">
        <string name="vendor">LG</string>
        <string name="model">CRD-8521B</string>
        <string name="type">cd</string>
      </nlist>
      <nlist name="eth0" type="record">
        <string name="vendor">3Com</string>
        <string name="model">3c905B-Combo [Deluxe Etherlink XL 10/100]</string>
        <string name="type">net</string>
        <string name="driver">3c59x</string>
        <string name="address">00:0d:0b:7:a9:a3:47</string>
      </nlist>
    </nlist>
  </nlist>
  <nlist name="system" type="record">
    <list name="mounts">
      <nlist type="record">
        <string name="type">swap</string>
        <string name="path">swap</string>
        <string name="device">hdal</string>
      </nlist>
      <nlist type="record">
        <string name="type">ext2</string>
        <string name="path">/</string>
        <string name="device">hda2</string>
      </nlist>
      <nlist type="record">
        <string name="type">ext2</string>
        <string name="path">/var</string>
        <string name="device">hda3</string>
      </nlist>
      <nlist type="record">
        <string name="type">proc</string>
        <string name="path">/proc</string>
      </nlist>
      <nlist type="record">
        <string name="type">devpts</string>
        <string name="path">/dev/pts</string>
        <list name="options">
          <string value="nosuid" />
        </list>
      </nlist>
    </list>
  </nlist>
</profile>

```

```

<string>gid=5</string>
<string>mode=620</string>
</list>
</nlist>
<nlist type="record">
<string name="type">ext2</string>
<string name="path">/mnt/floppy</string>
<list name="options">
<string>noauto</string>
<string>owner</string>
</list>
<string name="device">fd0</string>
</nlist>
<nlist type="record">
<string name="type">afs</string>
<string name="path">/afs</string>
</nlist>
<nlist type="record">
<string name="type">iso9660</string>
<string name="path">/mnt/cdrom</string>
<list name="options">
<string>noauto</string>
<string>owner</string>
<string>ro</string>
</list>
<string name="device">hdc</string>
</nlist>
</list>
<nlist name="partitions" type="table">
<nlist name="hdal" type="record">
<string name="type">primary</string>
<string name="disk">hda</string>
<long name="size">512</long>
<long name="id">82</long>
</nlist>
<nlist name="hda2" type="record">
<string name="type">primary</string>
<string name="disk">hda</string>
<long name="size">18828</long>
<long name="id">83</long>
</nlist>
<nlist name="hda3" type="record">
<string name="type">primary</string>
<string name="disk">hda</string>
<long name="size">256</long>
<long name="id">83</long>
</nlist>
</nlist>
</nlist>
</nlist>

```

Appendix B: Partial HLD Examples

These HLD templates have been used to generate the LLD found in Appendix A. More sample code can be found on our web site [22].

functions.tpl

```

#####
# Useful functions.
#####

```

```

declaration template functions;
# insert_after(string, string, list): insert the first string after the second
# one (if found) or at the end (otherwise); the last argument is modified but
# also returned as the result of the function
define function insert_after = {
    if (argc != 3 ||
        !is_string(argv[0]) || !is_string(argv[1]) || !is_list(argv[2]))
        error("usage: insert_after(string, string, list)");
    idx = index(argv[1], argv[2]);
    if (idx < 0) {
        # not found, we insert at the end
        splice(argv[2], length(argv[2]), 0, list(argv[0]));
    } else {
        # found, we insert just after
        splice(argv[2], idx+1, 0, list(argv[0]));
    };
    return(argv[2]);
};

# given a disk name, return a table of three primary partitions for swap, root
# and /var with a very simple space allocation algorithm
define function simple_partitions = {
    if (argc != 1 || !is_string(argv[0]))
        error("usage: simple_partitions(string)");
    disk = argv[0];
    disk_size = value("/hardware/devices/" + disk + "/size");
    # swap is twice the size of the physical memory
    swap = nlist(
        "disk", disk,
        "type", "primary",
        "size", 2 * value("/hardware/memory/size"),
        "id", 82, # Linux swap
    );
    # var is 256MB for disks larger than 2GB, 128MB otherwise
    var = nlist(
        "disk", disk,
        "type", "primary",
        "size", if (disk_size > 2048) 256 else 128,
        "id", 83, # Linux
    );
    # root is the rest
    root = nlist(
        "disk", disk,
        "type", "primary",
        "size", disk_size - swap["size"] - var["size"],
        "id", 83, # Linux
    );
    # order of partitions is swap, root and var
    return(nlist(
        disk+"1", swap,
        disk+"2", root,
        disk+"3", var,
    ));
};

types.tpl
#####
# Useful (but simplified) types.
#####

```

```

declaration template types;
#####
# simple types
# unsigned long
#(old style) define type ulong = long with self >= 0;
define type ulong = long(0..);
# unsigned double
define type udouble = double(0..);
# IPv4 address in dotted number notation
define type ipv4 = string with {
    result = matches(self, '^(\d+)\.(\d+)\.(\d+)\.(\d+)$');
    if (length(result) == 0)
        return("bad string");
    i = 1;
    while (i <= 4) {
        x = to_long(result[i]);
        if (x > 255)
            return("chunk " + to_string(i) + " too big: " + result[i]);
        i = i + 1;
    };
    return(true);
};

#####
# hardware types
# memory record
define type memory_t = {
    "size" : ulong
};

# CPU record
define type cpu_t = {
    "vendor" : string
    "model" : string
    "speed" : udouble
};

# device record (describing some hardware devices such as disks)
define type device_t = {
    "type" : string with match(self, '^disk|cd|net$')
    "vendor" : string
    "model" : string
    "size" ? ulong
    "driver" ? string
    "address" ? string
};

# hardware record (describing some complete hardware information)
define type hardware_t = {
    "vendor" : string
    "model" : string
    "serial" : string
    "memory" : memory_t
    "cpus" : cpu_t[1..]    # list of at least one CPU
    "devices" : device_t{} # table of devices, indexed by names such as hda
};

#####
# system types
# mount record (describing what will end up in /etc/fstab)

```

```

define type mount_t = {
    "device" ? string
    "path" : string
    "type" : string
    "name" ? string
    "options" ? string[]
};

# partition record (describing how to partition the disks)
define type partition_t = {
    "disk" : string with value("/hardware/devices/"+self+"/type") == "disk"
    "type" : string
    "size" : ulong
    "id" : ulong
};

# system record (describing some of the system configuration)
define type system_t = {
    "mounts" : mount_t[...] # list of at least one mount
    "partitions" ? partition_t{} # table of partitions, indexed by, e.g., hdal
};

#####
# root type
# type of the root of the configuration information
define type root_t = {
    "hardware" : hardware_t # hardware subtree
    "system" : system_t # system subtree
};

# declare that root is indeed of the root type
type "/" = root_t;

```

hardware.tpl

```

#####
# Sample hardware data.
#####
# cpus
structure template cpu_intel_p3_800;
"vendor" = "Intel";
"model" = "Pentium III (Coppermine)";
"speed" = 796.550; # MHz

structure template cpu_intel_p3_850;
"vendor" = "Intel";
"model" = "Pentium III (Coppermine)";
"speed" = 853.220; # MHz

#####
# disks
structure template disk_quantum_fireballp_as20_5;
"type" = "disk";
"vendor" = "QUANTUM";
"model" = "FIREBALLP AS20.5";
"size" = 19596; # MB

structure template disk_ibm_dtla_307030;
"type" = "disk";
"vendor" = "IBM";
"model" = "DTLA-307030";
"size" = 29314; # MB

```

```
#####
# cdroms
#####
structure template cdrom_lg_crd_8521b;
"type"      = "cd";
"vendor"    = "LG";
"model"     = "CRD-8521B";
#####

#####
# network cards
#####
structure template network_3com_3c905b;
"type"      = "net";
"vendor"    = "3Com";
"model"     = "3c905B-Combo [Deluxe Etherlink XL 10/100]";
"driver"   = "3c59x";

structure template network_intel_82557;
"type"      = "net";
"vendor"    = "Intel";
"model"     = "82557 [Ethernet Pro 100]";
"driver"   = "eepro100";
#####

#####
# computers
#####
structure template pc_elonex_850_256;
"vendor"    = "Elonex";
"model"     = "850/256";
"cpus"      = list(create("cpu_intel_p3_850"));
"memory/size" = 256; # MB
"devices/hda" = create("disk_quantum_fireballp_as20_5");
"devices/hdc" = create("cdrom_lg_crd_8521b");
"devices/eth0" = create("network_3com_3c905b");

structure template pc_elonex_800x2_512;
"vendor"    = "Elonex";
"model"     = "800x2/512";
"cpus"      = list(create("cpu_intel_p3_800"), create("cpu_intel_p3_800"));
"memory/size" = 512; # MB
"devices/hda" = create("disk_ibm_dtla_307030");
"devices/eth0" = create("network_intel_82557");

system.tpl
#####
# Sample system data.
#####
#####

#####
# standard mounts
#####
structure template mount_afs;
"path"      = "/afs";
"type"     = "afs";

structure template mount_proc;
"path"      = "/proc";
"type"     = "proc";

structure template mount_devpts;
"path"      = "/dev/pts";
"type"     = "devpts";
"options"  = list("gid=5", "mode=620");
#####
```

```

structure template mount_floppy;
"device" = "fd0";
"path" = "/mnt/floppy";
"type" = "ext2";
"options" = list("noauto", "owner");

structure template mount_cdrom;
"device" = undef;
"path" = "/mnt/cdrom";
"type" = "iso9660";
"options" = list("noauto", "owner", "ro");
#####
# mounting templates

# add the standard Linux mount entries
template mounting_linux;
"/system-mounts" = merge(value("/system-mounts"), list(
  create("mount_proc"),
  create("mount_devpts"),
  create("mount_floppy"),
));
# add the AFS mount entry
template mounting_afs;
"/system-mounts" = merge(value("/system-mounts"), list(create("mount_afs")));

sample.tpl
#####
# Sample object template.
#####
object template sample;
# standard includes
include types;
include functions;
# hardware information
"/hardware" = create("pc_elonex_850_256");
"/hardware/serial" = "CH01112041";
"/hardware/devices/eth0/address" = "00:d0:b7:a9:a3:47";
# system information
"/system-partitions" = simple_partitions("hda");
"/system-mounts/0" = nlist("type", "swap", "path", "swap", "device", "hda1");
"/system-mounts/1" = nlist("type", "ext2", "path", "/", "device", "hda2");
"/system-mounts/2" = nlist("type", "ext2", "path", "/var", "device", "hda3");
include mounting_linux;
include mounting_afs;
# we also add a mount entry for our CD drive ...
"/system-mounts" = merge(value("/system-mounts"),
  list(create("mount_cdrom", "device", "hdc")));
# ...and make sure that hdc indeed contains a CD drive!
valid "/hardware/devices/hdc" = self["type"] == "cd";

```

Appendix C: NFS Validation Example

xvalidation.tpl

```

#####
# Simplified example of cross object validation.
# All the NFS clients check that the NFS servers that they use indeed export
# the directories that they mount. This is done transparently by adding some

```

```

# validation code to the mount record type. Further checks such as wildcards
# in export list or export/mount options mismatch are left as an exercise for
# the reader ;-)

# Here is how to compile the server and two clients (result on stdout):
# % pan --stdout --output=nfssrv1 xvalidation.tpl                      (will succeed)
# % pan --stdout --output=nfscl1 xvalidation.tpl                      (will succeed)
# % pan --stdout --output=nfscl2 xvalidation.tpl                      (will fail)
#####
#####

# types definitions

template types:

# export record (roughly what is in /etc/exports)
define type export = {
    "path"   : string  description "path of the exported directory"
    "client" : string  description "name of client allowed to mount it"
    "options" ? string[] description "list of exporting options like ro"
};

# mount record (roughly what is in /etc/fstab)
define type mount = {
    "device" : string  description "device as understood by the mount command"
    "path"   : string  description "path of the mount point"
    "type"   : string  description "type of the mounted filesystem"
    "name"   ? string  description "name or label of this mount entry"
    "options" ? string[] description "list of mounting options like ro"
} with valid_mount(self);

# validation of a mount record (only nfs type records are checked)
define function valid_mount = {
    # the mount record is our only argument
    mount = argv[0];
    # we only care about NFS mounts, other types are considered OK
    if (mount["type"] != "nfs")
        return(true);
    # the device field will give us the NFS server and path
    result = matches(mount["device"], '^([\w\.\-]+)(.+)$');
    if (length(result) == 0)
        error("bad nfs device: " + mount["device"]);
    server = result[1];
    path   = result[2];
    # we now look at the server's exports list
    exports = value("//" + server + "/system/exports");
    i = 0;
    len = length(exports);
    while (i < len) {
        # we check if this export record is good for us by checking the client
        # field against object (i.e., the name of the current object template)
        # and the path: we want exact match and ignore the export/mount options
        if (exports[i]["client"] == object && exports[i]["path"] == path)
            return(true);
        i = i + 1;
    };
    # we haven't found any export record matching our needs, we complain:
    error("server " + server + " does not export " + path + " to " + object);
};

#####
#####

# NFS server definition

```

```
object template nfssrv1;
# type settings
include types;
type "/system/exports" = export[];
# data for this host
"/system/exports" = list(
    nlist(                                # we export /home to hostx
        "path",      "/home",
        "client",    "hostx",
    ),
    nlist(                                # we export /home to nfsclt1, read-only
        "path",      "/home",
        "client",    "nfsclt1",
        "options",   list("ro"),
    ),
);
#####
# NFS clients definitions

template client;
# type settings
include types;
type "/system/mounts" = mount[];
# data for this host
"/system/mounts" = list(
    nlist(                                # we mount /dev/hdal as the root filesystem
        "device",   "/dev/hdal",
        "path",     "/",
        "type",     "ext2",
    ),
    nlist(                                # we NFS mount /home from the server nfssrv1
        "device",   "nfssrv1:/home",
        "path",     "/home",
        "type",     "nfs",
    ),
);
#
# first client: known by the server, compilation will succeed
object template nfsclt1;
include client;

# second client: unknown to the server, compilation will fail with:
# *** user error: server nfssrv1 does not export /home to nfsclt2
object template nfsclt2;
include client;
```

Why Order Matters: Turing Equivalence in Automated Systems Administration

*Steve Traugott – TerraLuna, LLC
Lance Brown – National Institute of Environmental Health Sciences*

ABSTRACT

Hosts in a well-architected enterprise infrastructure are self-administered; they perform their own maintenance and upgrades. By definition, self-administered hosts execute self-modifying code. They do not behave according to simple state machine rules, but can incorporate complex feedback loops and evolutionary recursion.

The implications of this behavior are of immediate concern to the reliability, security, and ownership costs of enterprise and mission-critical computing. In retrospect, it appears that the same concerns also apply to manually-administered machines, in which administrators use tools that execute in the context of the target disk to change the contents of the same disk. The self-modifying behavior of both manual and automatic administration techniques helps explain the difficulty and expense of maintaining high availability and security in conventionally-administered infrastructures.

The practice of infrastructure architecture tool design exists to bring order to this self-referential chaos. Conventional systems administration can be greatly improved upon through discipline, culture, and adoption of practices better fitted to enterprise needs. Creating a low-cost maintenance strategy largely remains an art. What can we do to put this art into the hands of relatively junior administrators? We think that part of the answer includes adopting a well-proven strategy for maintenance tools, based in part upon the theoretical properties of computing.

In this paper, we equate self-administered hosts to Turing machines in order to help build a theoretical foundation for understanding this behavior. We discuss some tools that provide mechanisms for reliably managing self-administered hosts, using deterministic ordering techniques.

Based on our findings, it appears that no tool, written in any language, can predictably administer an enterprise infrastructure without maintaining a deterministic, repeatable order of changes on each host. The runtime environment for any tool always executes in the context of the target operating system; changes can affect the behavior of the tool itself, creating circular dependencies. The behavior of these changes may be difficult to predict in advance, so testing is necessary to validate changed hosts. Once changes have been validated in testing they must be replicated in production in the same order in which they were tested, due to these same circular dependencies.

The least-cost method of managing multiple hosts also appears to be deterministic ordering. All other known management methods seem to include either more testing or higher risk for each host managed.

This paper is a living document; revisions and discussion can be found at Infrastructures.Org, a project of TerraLuna, LLC.

Foreword *by Steve Traugott*

In 1998, Joel Huddleston and I suggested that an entire enterprise infrastructure could be managed as one large “enterprise virtual machine” (EVM) [bootstrap]. That paper briefly described parts of a management toolset, later named ISconf [isconf]. This toolset, based on relatively simple makefiles and shell scripts, did not seem extraordinary at the time. At one point in the paper, we said that we would likely use cfengine [cfengine] the next time around – I had been following Mark Burgess’ progress since 1994.

That 1998 paper spawned a web site and community at Infrastructures.Org. This community in turn helped launch the Infrastructure Architecture (IA) career field. In the intervening years, we’ve seen the Infrastructures.Org community grow from a few dozen to a few hundred people, and the IA field blossom from obscurity into a major marketing campaign by a leading systems vendor.

Since 1998, Joel and I have both attempted to use other tools, including cfengine version 1. I’ve also tried to write tools from scratch again several times, with mixed success. We have repeatedly hit

indications that our 1998 toolset was more optimized than we had originally thought. It appears that in some ways Joel and I, and the rest of our group at the Bank, were lucky; our toolset protected us from many of the pitfalls that are laying in wait for IAs.

One of these pitfalls appears to be deterministic ordering; I never realized how important it was until I tried to use other tools that don't support it. When left without the ability to concisely describe the order of changes to be made on a machine, I've seen a marked decrease in my ability to predict the behavior of those changes, and a large increase in my own time spent monitoring, troubleshooting, and coding for exceptions. These experiences have shown me that loss of order seems to result in lower production reliability and higher labor cost.

The ordered behavior of ISconf was more by accident than design. I needed a quick way to get a grip on 300 machines. I cobbled a prototype together on my HP100LX palmtop one March '94 morning, during the 35-minute train ride into Manhattan. I used 'make' as the state engine because it's available on most UNIX machines. The deterministic behavior 'make' uses when iterating over prerequisite lists is something I didn't think of as important at the time – I was more concerned with observing known dependencies than creating repeatable order.

Using that toolset and the EVM mindset, we were able to repeatedly respond to the chaotic international banking mergers and acquisitions of the mid-90's. This response included building and rebuilding some of the largest trading floors in the world, launching on schedule each time, often with as little as a few months' notice, each launch cleaner than the last. We knew at the time that these projects were difficult; after trying other tool combinations for more recent projects I think I have a better appreciation for just how difficult they were. The phrase "throwing a truck through the eye of a needle" has crossed my mind more than once. I don't think we even knew the needle was there.

At the invitation of Mark Burgess, I joined his LISA 2001 [lisa] cfengine workshop to discuss what we'd found so far, with possible targets for the cfengine 2.0 feature set. The ordering requirement seemed to need more work; I found ordering surprisingly difficult to justify to an audience practiced in the use of convergent tools, where ordering is often considered a constraint to be specifically avoided [couch, eika-sandnes]. Later that week, Lance Brown and I were discussing this over dinner, and he hit on the idea of comparing a UNIX machine to a Turing machine. The result is this paper.

Based on the symptoms we have seen when comparing ISconf to other tools, I suspect that ordering is a keystone principle in automated systems administration. Lance and I, with a lot of help from others, will attempt to offer a theoretical basis for this suspicion.

We encourage others to attempt to refute or support this work at will; I think systems administration may be about to find its computer science roots. We have also already accumulated a large FAQ for this paper – we'll put that on the website. Discussion on this paper as well as related topics is encouraged on the *infrastructures* mailing list at <http://Infrastructures.Org>.

Why Order Matters

There seem to be (at least) several major reasons why the order of changes made to machines is important in the administration of an enterprise infrastructure:

A "circular dependency" or control-loop problem exists when an administrative tool executes code that modifies the tool or the tool's own foundations (the underlying host). Automated administration tool designers cannot assume that the users of their tool will always understand the complex behavior of these circular dependencies. In most cases we will never know what dependencies end users might create; see assertions §A.40 and §A.46 in the 'Turing Equivalence' section of this paper.

A test infrastructure is needed to test the behavior of changes before rolling them to production. No tool or language can remove this need, because no testing is capable of validating a change in any conditions other than those tested. This test infrastructure is useless unless there is a way to ensure that production machines will be built and modified in the same way as the test machines; see 'The Need for Testing' section.

It appears that a tool that produces deterministic order of changes is cheaper to use than one that permits more flexible ordering. The unpredictable behavior resulting from unordered changes to disk is more costly to validate than the predictable behavior produced by deterministic ordering; see §A.58. Because cost is a significant driver in the decision-making process of most IT organizations, we will discuss this point more in the next section.

Local staff must be able to use administrative tools after a cost-effective (i.e., cheap and quick) turnover phase. While senior infrastructure architects may be well-versed in avoiding the pitfalls of unordered change, we cannot be on the permanent staff of every IT shop on the globe. In order to ensure continued health of machines after rollout of our tools, the tools themselves need to have some reasonable default behavior that is safe if the user lacks this theoretical knowledge; see §A.40 and §A.54.

This business requirement must be addressed by tool developers. In our own practice, we have been able to successfully turnover enterprise infrastructures to permanent staff many times over the last several years. Turnover training in our case is relatively simple, because our toolsets have always implemented ordered change by default. Without this default behavior, we would have also needed to attempt to teach

advanced techniques needed for dealing with unordered behavior, such as inspection of code in vendor-supplied binary packages; see the ‘Right Packages, Wrong Order’ section.

A Prediction

“Order Matters” when we care about both quality and cost while maintaining an enterprise infrastructure. If the ideas described in this paper are correct, then we can make the following prediction:

The least-cost way to ensure that the behavior of any two hosts will remain completely identical is always to implement the same changes in the same order on both hosts.

This sounds very simple, almost intuitive, and for many people it is. But to our knowledge, `isconf` [`isconf`] is the only generally-available tool which specifically supports administering hosts this way. There seems to be no prior art describing this principle, and in our own experience we have yet to see it specified in any operational procedure. It is trivially easy to demonstrate in practice, but has at times been surprisingly hard to support in conversation, due to the complexity of theory required for a proof.

Note that this prediction does not apply only to those situations when you want to maintain two or more identical hosts. It applies to any computer-using organization that needs cost-effective, reliable operation. This includes those that have many unique production hosts; see ‘The Need for Testing.’ The ‘Congruence’ section discusses this further, including single-host rebuilds after a security breach.

This prediction also applies to disaster recovery (DR) or business continuity planning. Any part of a credible DR procedure includes some method of rebuilding lost hosts, often with new hardware, in a new location. Restoring from backups is one way to do this, but making complete backups of multiple hosts is redundant – the same operating system components must be backed up for each host, when all we really need are the user data and host build procedures (how many copies of `/bin/ls` do we really need on tape?). It is usually more efficient to have a means to quickly and correctly rebuild each host from scratch. A tool that maintains an ordered record of changes made after install is one way to do this.

This prediction is particularly important for those organizations using what we call *self-administered hosts*. These are hosts that run an automated configuration or administration tool in the context of their own operating environment. Commercial tools in this category include Tivoli, Opsware, and CenterRun [tivoli, opsware, centerrun]. Open-source tools include `cfengine`, `lcfg`, `pikt`, and our own `isconf` [`cfengine`, `lcfg`, `pikt`, `isconf`]. We will discuss the fitness of some of these tools later – not all appear fully suited to the task.

This prediction applies to those organizations which still use an older practice called “cloning” to

create and manage hosts. In cloning, an administrator or tool copies a disk image from one machine to another, then makes the changes needed to make the host unique (at minimum, IP address and hostname). After these initial changes, the administrator will often make further changes over the life of the machine. These changes may be required for additional functionality or security, but are too minor to justify re-cloning. Unless order is observed, identical changes made to multiple hosts are not guaranteed to behave in a predictable way (§A.47). The procedure needed for properly maintaining cloned machines is not substantially different from that described in the section on ‘Describing Disk State.’

This prediction, stated more formally in §A.58, seems to apply to UNIX, Windows, and any other general-purpose computer with a rewritable disk and modern operating system. More generally, it seems to apply to any von Neumann machine with rewritable nonvolatile storage.

Management Methods

All computer systems management methods can be classified into one of three categories: divergent, convergent, and congruent.

Divergence

Divergence (Figure 1) generally implies bad management. Experience shows us that virtually all enterprise infrastructures are still divergent today. Divergence is characterized by the configuration of live hosts drifting away from any desired or assumed baseline disk content.

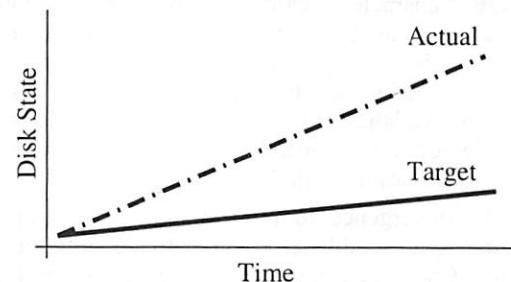


Figure 1: Divergence.

One quick way to tell if a shop is divergent is to ask how changes are made on production hosts, how those same changes are incorporated into the baseline build for new or replacement hosts, and how they are made on hosts that were down at the time the change was first deployed. If you get different answers, then the shop is likely divergent.

The symptoms of divergence include unpredictable host behavior, unscheduled downtime, unexpected package and patch installation failure, unclosed security vulnerabilities, significant time spent “firefighting,” and high troubleshooting and maintenance costs.

The causes of divergence are generally that class of operations that create non-reproducible change.

Divergence can be caused by ad hoc manual changes, changes implemented by two independent automatic agents on the same host, and other unordered changes. Scripts which drive rdist, rsync, ssh, scp, [rdist, rsync, ssh] or other change agents as a push operation [bootstrap] are also a common source of divergence.

Convergence

Convergence (Figure 2) is the process most senior systems administrators first begin when presented with a divergent infrastructure. They tend to start by manually synchronizing some critical files across the diverged machines, then they figure out a way to do that automatically. Convergence is characterized by the configuration of live hosts moving towards an ideal baseline. By definition, all converging infrastructures are still diverged to some degree. (If an infrastructure maintains full compliance with a fully descriptive baseline, then it is congruent according to our definition, not convergent; see the ‘Congruence’ section.)

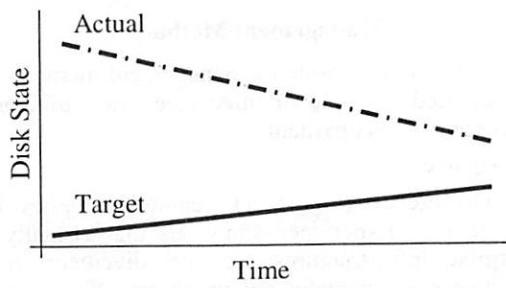


Figure 2: Convergence.

The baseline description in a converging infrastructure is characteristically an incomplete description of machine state. You can quickly detect convergence in a shop by asking how many files are currently under management control. If an approximate answer is readily available and is on the order of a few hundred files or less, then the shop is likely converging legacy machines on a file-by-file basis.

A convergence tool is an excellent means of bringing some semblance of order to a chaotic infrastructure. Convergent tools typically work by sampling a small subset of the disk – via a checksum of one or more files, for example – and taking some action in response to what they find. The samples and actions are often defined in a declarative or descriptive language that is optimized for this use. This emulates and preempts the firefighting behavior of a reactive human systems administrator – “see a problem, fix it.” Automating this process provides great economies of scale and speed over doing the same thing manually.

Convergence is a feature of Mark Burgess’ Computer Immunology principles [immunology]. His cfengine is in our opinion the best tool for this job [cfengine]. Simple file replication tools [sup, cvsup, rsync] provide a rudimentary convergence function, but without the other action semantics and fine-grained control that cfengine provides.

Because convergence typically includes an intentional process of managing a specific subset of files, there will always be unmanaged files on each host. Whether current differences between unmanaged files will have an impact on future changes is undecidable, because at any point in time we do not know the entire set of future changes, or what files they will depend on.

It appears that a central problem with convergent administration of an initially divergent infrastructure is that there is no documentation or knowledge as to when convergence is complete. One must treat the whole infrastructure as if the convergence is incomplete, whether it is or not. So without more information, an attempt to converge formerly divergent hosts to an ideal configuration is a never-ending process. By contrast, an infrastructure based upon first loading a known baseline configuration on all hosts, and limited to purely orthogonal and non-interacting sets of changes, implements congruence (defined in the next section). Unfortunately, this is not the way most shops use convergent tools such as cfengine.

The symptoms of a convergent infrastructure include a need to test all changes on all production hosts, in order to detect failures caused by remaining unforeseen differences between hosts. These failures can impact production availability. The deployment process includes iterative adjustment of the configuration tools in response to newly discovered differences, which can cause unexpected delays when rolling out new packages or changes. There may be a higher incidence of failures when deploying changes to older hosts. There may be difficulty eliminating some of the last vestiges of the ad-hoc methods mentioned in the section on ‘Divergence.’ Continued use of ad-hoc and manual methods virtually ensures that convergence cannot complete.

With all of these faults, convergence still provides much lower overall maintenance costs and better reliability than what is available in a divergent infrastructure. Convergence features also provide more adaptive self-healing ability than pure congruence, due to a convergence tool’s ability to detect when deviations from baseline have occurred. Congruent infrastructures rely on monitoring to detect deviations, and generally call for a rebuild when they have occurred. We discuss the security reasons for this in the ‘Congruence’ section.

We have found apparent limits to how far convergence alone can go. We know of no previously divergent infrastructure that, through convergence alone, has reached congruence. This makes sense; convergence is a process of eliminating differences on an as-needed basis; the managed disk content will generally be a smaller set than the unmanaged content. In order to prove congruence, we would need to sample all bits on each disk, ignore those that are user data, determine which of the remaining bits are relevant to the operation of the machine, and compare those with the baseline.

In our experience, it is not enough to prove via testing that two hosts currently exhibit the same behavior while ignoring bit differences on disk; we care not only about current behavior, but future behavior as well. Bit differences that are currently deemed not functional, or even those that truly have not been exercised in the operation of the machine, may still affect the viability of future change directives. If we cannot predict the viability of future change actions, we cannot predict the future viability of the machine.

Deciding what bit differences are “functional” is often open to individual interpretation. For instance, do we care about the order of lines and comments in /etc/inetd.conf? We might strip out comments and reorder lines without affecting the current operation of the machine; this might seem like a non-functional change, until two years from now. After time passes, the lack of comments will affect our future ability to correctly understand the infrastructure when designing a new change. This example would seem to indicate that even non-machine-readable bit differences can be meaningful when attempting to prove congruence.

Unless we can prove congruence, we cannot validate the fitness of a machine without thorough testing, due to the uncertainties described in §A.25. In order to be valid, this testing must be performed on each production host, due to the factors described in §A.47. This testing itself requires either removing the host from production use or exposing untested code to users. Without this validation, we cannot trust the machine in mission-critical operation.

Congruence

Congruence (Figure 3) is the practice of maintaining production hosts in complete compliance with a fully descriptive baseline (see the section on ‘Describing Disk State’). Congruence is defined in terms of disk state rather than behavior, because disk state can be fully described, while behavior cannot (§A.59).

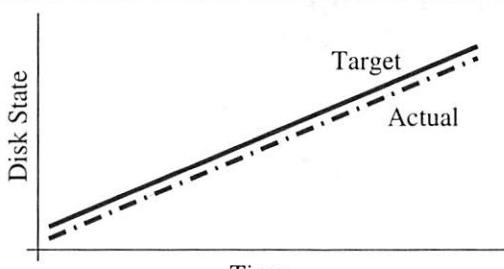


Figure 3: Congruence.

By definition, divergence from baseline disk state in a congruent environment is symptomatic of a failure of code, administrative procedures, or security. In any of these three cases, we may not be able to assume that we know exactly which disk content was damaged. It is usually safe to handle all three cases as a security breach: correct the root cause, then rebuild.

You can detect congruence in a shop by asking how the oldest, most complex machine in the infrastructure would be rebuilt if destroyed. If years of sysadmin work can be replayed in an hour, unattended, without resorting to backups, and only user data need be restored from tape, then host management is likely congruent.

Rebuilds in a congruent infrastructure are completely unattended and generally faster than in any other; anywhere from ten minutes for a simple workstation to two hours for a node in a complex high-availability server cluster (most of that two hours is spent in blocking sleeps while meeting barrier conditions with other nodes).

Symptoms of a congruent infrastructure include rapid, predictable, “fire-and-forget” deployments and changes. Disaster recovery and production sites can be easily maintained or rebuilt on demand in a bit-for-bit identical state. Changes are not tested for the first time in production, and there are no unforeseen differences between hosts. Unscheduled production downtime is reduced to that caused by hardware and application problems; firefighting activities drop considerably. Old and new hosts are equally predictable and maintainable, and there are fewer host classes to maintain. There are no ad-hoc or manual changes. We have found that congruence makes cost of ownership much lower, and reliability much higher, than any other method.

Our own experience and calculations show that the return-on-investment (ROI) of converting from divergence to congruence is less than 8 months for most organizations; see Figure 4. This graph assumes an existing divergent infrastructure of 300 hosts, 2%/month growth rate, followed by adoption of congruent automation techniques. Typical observed values were used for other input parameters. Automation tool rollout began at the 6-month mark in this graph, causing temporarily higher costs; return on this investment is in 5 months, where the manual and automatic lines cross over at the 11 month mark. Following crossover, we see a rapidly increasing cost savings, continuing over the life of the infrastructure. While this graph is calculated, the results agree with actual enterprise environments that we have converted. There is a CGI generator for this graph at Infrastructure.org, where you can experiment with your own parameters.

Congruence allows us to validate a change on one host in a class, in an expendable test environment, then deploy that change to production without risk of failure. Note that this is useful even when (or especially when) there may be only one production host in that class.

A congruence tool typically works by maintaining a journal of all changes to be made to each machine, including the initial image installation. The journal entries for a class of machine drive all changes on all machines in that class. The tool keeps a lifetime

record, on the machine's local disk, of all changes that have been made on a given machine. In the case of loss of a machine, all changes made can be recreated on a new machine by "replaying" the same journal; likewise for creating multiple, identical hosts. The journal is usually specified in a declarative language that is optimized for expressing ordered sets and subsets. This allows subclassing and easy reuse of code to create new host types; see 'Describing Disk State.'

There are few tools that are capable of the ordered lifetime journaling required for congruent behavior. Our own `isconf` (described in its own section) is the only specifically congruent tool we know of in production use, though `cfengine`, with some care and extra coding, appears to be usable for administration of congruent environments. We discuss this in more detail in the 'Cfengine Techniques' section.

We recognize that congruence may be the only acceptable technique for managing life-critical systems infrastructures, including those that:

- Influence the results of human-subject health and medicine experiments
- Provide command, control, communications, and intelligence (C^3I) for battlefield and weapons systems environments
- Support command and telemetry systems for manned aerospace vehicles, including spacecraft and national airspace air traffic control

Our personal experience shows that awareness of the risks of conventional host management techniques has not yet penetrated many of these organizations. This is cause for concern.

Ordered Thinking

We have found that designers of automated systems administration tools can benefit from a certain mindset:

Think like a kernel developer, not an application programmer.

A good multitasking operating system is designed to isolate applications (and their bugs) from each other and from the kernel, and produce the illusion of independent execution. Systems administration is all about making sure that users continue to see that illusion.

Modern languages, compilers, and operating systems are designed to isolate applications programmers from "the bare hardware" and the low-level machine code, and enable object-oriented, declarative, and other high-level abstractions. But it is important to remember that the central processing unit(s) on a general-purpose computer only accepts machine-code instructions, and these instructions are coded in a procedural language. High-level languages are convenient abstractions, but are dependent on several layers of code to deliver machine language instructions to the CPU.

In reality, on any computer there is only one program; it starts running when the machine finishes power-on self test (POST), and stops when you kill the power. This program is machine language code, dynamically linked at runtime, calling in fragments of code from all over the disk. These "fragments" of code are what we conventionally think of as applications, shared libraries, device drivers, scripts, commands, administrative tools, and the kernel itself – all of the components that make up the machine's operating environment.

None of these fragments can run standalone on the bare hardware – they all depend on others. We cannot analyze the behavior of any application-layer tool as if it were a standalone program. Even kernel startup depends on the bootloader, and in some operating systems the kernel runtime characteristics can be influenced by one or more configuration files found elsewhere on disk.

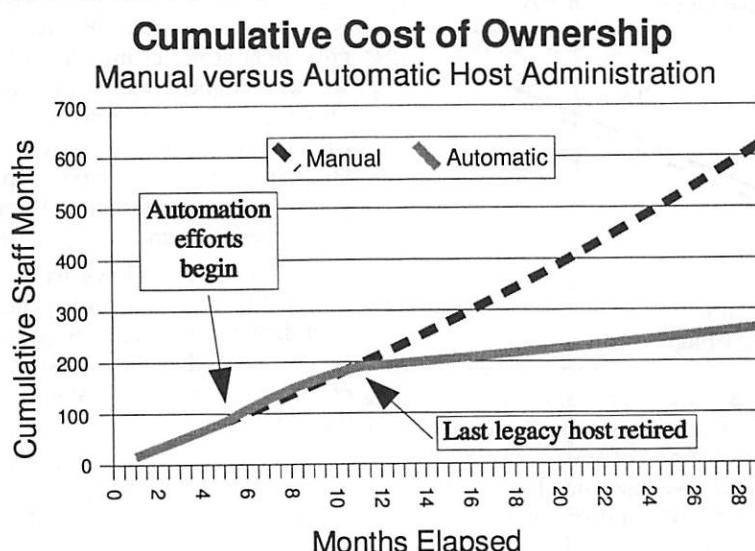


Figure 4: Cumulative costs for fully automated (congruent) versus manual administration.

This perspective is opposite from that of an application programmer. An application programmer “sees” the system as an axiomatic underlying support infrastructure, with the application in control, and the kernel and shared libraries providing resources. A kernel developer, though, is on the other side of the syscall interface; from this perspective, an application is something you load, schedule, confine, and kill if necessary.

On a UNIX machine, systems administration tools are generally ordinary applications that run as root. This means that they, too, are at the mercy of the kernel. The kernel controls them, not the other way around. And yet, we depend on automated systems administration tools to control, modify, and occasionally replace not only that kernel, but any and all other disk content. This presents us with the potential for a circular dependency chain.

A common misconception is that “there is some high-level tool language that will avoid the need to maintain strict ordering of changes on a UNIX machine.” This belief requires that the underlying runtime layers obey axiomatic and immutable behavioral laws. When using automated administration tools we cannot consider the underlying layers to be axiomatic; the administration tool itself perturbs those underlying layers; see the ‘Circular Dependencies’ section.

Inspection of high-level code alone is not enough. Without considering the entire system and its resulting machine language code, we cannot prove correctness. For example:

```
print "hello\n";
```

This looks like a trivial-enough Perl program; it “obviously” should work. But what if the Perl interpreter is broken? In other words, a conclusion of “simple enough to easily prove” can only be made by analyzing low-level machine language code, and the means by which it is produced.

“Order Matters” because we need to ensure that the machine-language instructions resulting from a set of change actions will execute in the correct order, with the correct operands. Unless we can prove program correctness at this low level, we cannot prove the correctness of any program. It does no good to prove correctness of a higher-level program when we do not know the correctness of the lower runtime layers. If the high-level program can modify those underlying layers, then the behavior of the program can change with each modification. Ordering of those modifications appears to be important to our ability to predict the behavior of the high-level program. (Put simply, it is important to ensure that you can step off of the tree limb *before* you cut through it.)

The Need for Testing

Just as we urge tool designers to think like kernel developers, we urge systems administrators to think

like operating systems vendors – because they are. Systems administration is actually *systems modification*; the administrator replaces binaries and alters configuration files, creating a combination which the operating system vendor has never tested. Since many of these modifications are specific to a single site or even a single machine, it is unreasonable to assume that the vendor has done the requisite testing. The systems administrator must perform the role of systems vendor, testing each unique combination – before the users do.

Due to modern society’s reliance on computers, it is unethical (and just plain bad business practice) for an operating system vendor to release untested operating systems without at least noting them as such. Better system vendors undertake a rigorous and exhaustive series of unit, system, regression, application, stress, and performance testing on each build before release, knowing full well that no amount of testing is ever enough (§A.9). They do this in their own labs; it would make little sense to plan to do this testing on customers’ production machines.

And yet, IT shops today habitually have no dedicated testing environment for validating changed operating systems. They deploy changes directly to production without prior testing. Our own experience and informal surveys show that greater than 95% of shops still do business this way. It is no wonder that reliability, security, and high availability are still major issues in IT.

We urge systems administrators to create and use dedicated testing environments (§A.42), not inflict changes on users without prior testing, and consider themselves the operating systems vendors that they really are. We urge IT management organizations to understand and support administrators in these efforts; the return on investment is in the form of lower labor costs and much higher user satisfaction. Availability of a test environment enables the deployment of automated systems administration tools, bringing major cost savings; see Figure 4.

A test environment is useless until we have a means to replicate the changes we made in testing onto production machines. “Order matters” when we do this replication; an earlier change will often affect the outcome of a later change. This means that changes made to a test machine must later be “replayed” in the same order on the machine’s production counterpart; see §A.45.

Testing costs can be greatly reduced by limiting the number of unique builds produced; this holds true for both vendors and administrators. This calls for careful management of changes and host classes in an IT environment, with an intent of limiting proliferation of classes; see §A.41.

Note that use of open-source operating systems does not remove the need for local testing of local modifications. In any reasonably complex infrastructure, there will always be local configuration and non-

packaged binary modifications which the community cannot have previously exercised. We prefer open source; we do not expect it to relieve us from our responsibilities though.

Ordering HOWTO

Automated systems administration is very straightforward. There is only one way for a user-side administrative tool to change the contents of disk in a running UNIX machine – the syscall interface. The task of automated administration is simply to make sure that each machine's kernel gets the right system calls, in the right order, to make it be the machine you want it to be.

Describing Disk State

If there are N bits on a disk, then there are 2^N possible disk states. In order to maintain the baseline host description needed for congruent management, we need to have a way to describe any arbitrary disk state in a highly compressed way, preferably in a human-readable configuration file or script. For the purposes of this description, we neglect user data and log files – we want to be able to describe the root-owned and administered portions of disk. “Order Matters” whether creating or modifying a disk:

A concise and reliable way to describe any arbitrary state of a disk is to describe the procedure for creating that state.

This procedure will include the initial state (bare-metal build) of the disk, followed by the steps used to change it over time, culminating in the desired state. This procedure must be in writing, preferably in machine-readable form. This entire set of information, for all hosts, constitutes the baseline description of a congruent infrastructure. Each change added to the procedure updates the baseline. See the ‘Congruence’ section.

There are tools which can help you maintain and execute this procedure. See the ‘Example Tools and Techniques’ section, particularly ‘Baseline Management in ISconf’.

While it is conceivable that this procedure could be a documented manual process, executing these steps manually is tedious and costly at best. (Though we know of many large mission-critical shops which try.) It is generally error-prone. Manual execution of complex procedures is one of the best methods we know of for generating divergence.

The starting state (bare-metal install) description of the disk may take the form of a network install tool's configuration file, such as that used for Solaris Jumpstart or RedHat Kickstart. The starting state might instead be a bitstream representing the entire initial content of the disk (usually a snapshot taken right after install from vendor CD). The choice of which of these methods to use is usually dependent on

the vendor-supplied install tool – some will support either method, some require one or the other.

How to Break an Enterprise

A systems administrator, whether a human or a piece of software (§A.36), can easily break an enterprise infrastructure by executing the right actions in the wrong order. In this section, we will explore some of the ways this can happen.

Right Commands, Wrong Order

First we will cover a trivial but devastating example that is easily avoided. This once happened to a colleague while doing manual operations on a machine. He wanted to clean out the contents of a directory which ordinarily had the development group's source code NFS mounted over top of it. Here is what he wanted to do:

```
umount /apps/src
cd /apps/src
rm -rf .
mount /apps/src
```

Here's what he actually did:

```
umount /apps/src
...
... umount fails, directory in use;
while resolving this, his pager goes
off, he handles the interrupt, then...
cd /apps/src
rm -rf .
```

Needless to say, there had also been no backup of the development source tree for quite some time...

In this example, “correct order” includes some concept of sufficient error handling. We show this example because it highlights the importance of a default behavior of “halt on error” for automatic systems administration tools. Not all tools halt on error by default; isconf does.

Right Packages, Wrong Order

We in the UNIX community have long accused Windows developers of poor library management, due to the fact that various Windows applications often come bundled with differing versions of the same DLLs. It turns out that at least some UNIX and Linux distributions appear to suffer from the same problem.

Jeffrey D'Amelia and John Hart [hart] demonstrated this in the case of RedHat RPMs, both official and contributed. They showed that the order in which you install RPMs can matter, even when there are no applicable dependencies specified in the package. We don't assume that this situation is restricted to RPMs only – any package management system should be susceptible to this problem. An interesting study would be to investigate similar overlaps in vendor-supplied packages for commercial UNIX distributions.

Detecting this problem for any set of packages involves extensive analysis by talented persons. In the case of [hart], the authors developed a suite of global

analysis tools, and repeatedly downloaded and unpacked thousands of RPMs. They still only saw “the tip of the iceberg” (their words). They intentionally ignored the actions of postinstall scripts, and they had not yet *executed* any packaged code to look for behavioral interactions.

Avoiding the problem is easier; install the packages, record the order of installation, test as usual, and when satisfied with testing, install the same packages in the same order on production machines.

While we’ve used packages in this example, we’d like to remind the reader that these considerations apply not only to package installation, but to any other change that affects the root-owned portions of disk.

Circular Dependencies

There is a “chicken and egg” or bootstrapping problem when updating either an automated systems administration tool (ASAT) or its underlying foundations (§A.40). Order is important when changes the tool makes can change the ability of the tool to make changes.

For example, cfengine version 2 includes new directives available for use in configuration files. Before using a new configuration file, the new version of cfengine needs to be installed. The new client is named ‘cfagent’ rather than ‘cfengine,’ so wrapper scripts and crontab entries should also be updated, and so on.

For fully automated operation on hundreds or thousands of machines, we would like to be able to upgrade cfengine under the control of cfengine (§A.46). We want to ensure that the following actions will take place on all machines, including those currently down:

1. fetch configuration file containing the following instructions
2. install new cfagent binary
3. run cfkey to generate key pair
4. fetch new configuration file containing version 2 directives
5. update calling scripts and crontab entries

There are several ordering considerations here. We won’t know that we need the new cfagent binary until we do step 1. We shouldn’t proceed with step 4 until we know that 2 and 3 were successful. If we do 5 too early, we may break the ability for cfengine to operate at all. If we do step 4 too early and try to run the resulting configuration file using the old version of cfengine, it will fail.

While this example may seem straightforward, implementing it in a language which does not by default support deterministic ordering requires much use of conditionals, state chaining, or equivalent. If this is the case, then code flow will not be readily apparent, making inspection and edits error-prone. Infrastructure automation code runs as root and has the ability to stop work across the entire enterprise; it needs to be simple,

short, and easy for humans to read, like security-related code paths in tools such as PGP or ssh.

If the tool’s language does not support “halt on error” by default, then it is easy to inadvertently allow later actions to take place when we would have preferred to abort. Going back to our cfengine example, if we can easily abort and leave the cfengine version 1 infrastructure in place, then we can still use version 1 to repair the damage.

Other Sources of Breakage

There are many other examples we could show, some including multi-host “barrier” problems. These include:

- Updating ssh to openssh on hundreds of hosts and getting the authorized_keys and/or protocol version configuration out of order. This can greatly hinder further contact with the target hosts. Daniel Hagerty [hagerty] ran into this one; many of us have been bitten by this at some point.
- Reconfiguring network routes or interfaces while communicating with the target device via those same routes or interfaces. Ordering errors can prevent further contact with the target, and often require a physical visit to resolve. This is especially true if the target is a workstation with no remote serial console access. Again, most readers have had this happen to them.

Example Tools and Techniques

While there are many automatic systems administration tools (ASAT) available, the two we are most familiar with are cfengine and our own isconf [cfengine, isconf]. In the next two sections, we will look at these two tools with a focus on how each can be used to create deterministic ordering.

In general, some of the techniques that seem to work well for the design and use of most ASATs include:

- Keep the “Turing tape” a finite size by holding the network content constant (§A.23), or versioning it using CVS or another version control tool [cvs, bootstrap]. This helps prevent some of the more insidious behaviors that are a potential in self-modifying machines (§A.40).
- Continuing in that vein, when using distributed package repositories such as the public Debian [debian] package server infrastructure, always specify version numbers when automating the installation of packages, rather than let the package installation tool (in Debian’s case apt-get) select the latest version. If you do not specify the package version, then you may introduce divergence. This risk varies, of course, depending on your choice of ‘stable’ or ‘unstable’ distribution, though we suspect it still applies in ‘stable,’ especially when using the ‘security’ packages. It certainly applies in all cases when you need to maintain your own

kernel or kernel modules rather than using the distributed packages.

We have experienced this repeatedly – machines which built correctly the first time with a given package list will not rebuild with the same package list a few weeks later, due to package version changes on the public servers, and resulting unresolved incompatibilities with local conditions and configuration file contents. Remember, your hosts are unique in the world – there are likely no others like them. Package maintainers cannot be expected to test every configuration, especially yours. You must retain this responsibility. See ‘The Need for Testing.’

We use Debian in this example because it is a distribution we like a lot; note that other package distribution and installation infrastructures, such as the RedHat up2date system, also have this problem.

- Expect long dependency or sequence chains when building enterprise infrastructures. If an ASAT can easily support encapsulation and ordering of 10, 50, or even 100 complex atomic actions in a single chain, then it is likely capable of fully automated administration of machines, including package, kernel, build, and even rebuild management. If the ASAT is cumbersome to use when chains become only two or three actions deep, then it is likely most suited for configuration file management, not package, binary, or kernel manipulation.

ISconf Techniques

As mentioned in the Foreword, isconf originally began life as a quick hack. Its basic utility has proven itself repeatedly over the last eight years, and as adoption has grown it is currently managing more production infrastructures than we are personally aware of.

While we show some ISconf makefile examples here, we do not show any example of the top-level configuration file which drives the environment and targets for ‘make.’ It is this top-level configuration file, and the scripts which interpret it, which are the core of ISconf and enable the typing or classing of hosts. These top-level facilities also are what govern the actions ISconf is to take during boot versus cron or other execution contexts. More information and code is available at ISconf.org and Infrastructures.Org.

We also do not show here the network fetch and update portions of ISconf, and the way that it updates its own code and configuration files at the beginning of each run. This default behavior is something that we feel is important in the design of any automated systems administration tool. If the tool does not support it, end-users will have to figure out how to do it safely themselves, reducing the usability of the tool.

ISconf Version 2

Version 2 of ISconf was a late-90’s rewrite to clean up and make portable the lessons learned from

version 1. As in version 1, the code used was Bourne shell, and the state engine used was ‘make.’

In Listing 1, we show a simplified example of Version 2 usage. While examples related to this can be found in [hart] and in our own makefiles, real-world usage is usually much more complex than the example shown here. We’ve contrived this one for clarity of explanation.

In this contrived example, we install two packages which we have not proven orthogonal. We in fact do not wish to take the time to detect whether or not they are orthogonal, due to the considerations expressed in §A.58. We may be tool users, rather than tool designers, and may not have the skillset to determine orthogonality, as in §A.54.

These packages might both affect the same shared library, for instance. Again according to [hart] and our own experience, it is not unusual for two packages such as these to list neither as prerequisites, so we might gain no ordering guidance from the package headers either.

In other words, all we know is that we installed package ‘foo,’ tested and deployed it to production, and then later installed package ‘bar,’ tested it and deployed. These installs may have been weeks or months apart. All went well throughout, users were happy, and we have no interest in unpacking and analyzing the contents of these packages for possible reordering for any reason; we’ve gone on to other problems.

Because we know this order works, we wish for these two packages, ‘foo’ and ‘bar,’ to be installed in the same order on every future machine in this class. This makefile will ensure that; make always iterates over a prerequisite list in the same order.

The touch \$@ command at the end of each stanza will prevent this stanza from being run again. The ISconf code always changes to the timestamps directory before starting ‘make’ (and takes other measures to constrain the normal behavior of ‘make,’ so that we never try to “rebuild” this target either).

The class name in this case (Listing 1) is ‘Block12.’ You can see that ‘Block12’ is also made up of many other packages; we don’t show the makefile stanzas for these here. These packages are listed as prerequisites to ‘Block12,’ in chronological order. Note that we only want to add items to the end of this list, not the middle, due to the considerations expressed in section §A.49.

In this example, even though we take advantage of the Debian package server infrastructure, we specify the version of package that we want, as in the introduction to the ‘Example Tools and Techniques’ section. We also use a caching proxy when fetching Debian packages, in order to speed up our own builds and reduce the load on the Debian servers to a minimum.

Note that we get “halt-on-error” behavior from ‘make,’ as we wished for in ‘Right commands, Wrong

Order.’ If any of the commands in the ‘foo’ or ‘bar’ sections exit with a non-zero return code, then ‘make’ aborts processing immediately. The ‘touch’ will not happen, and we normally configure the infrastructure such that the ISconf failure will be noticed by a monitoring tool and escalated for resolution. In practice, these failures very rarely occur in production; we see and fix them in test. Production failures, by the definition of congruence, usually indicate a systemic, security, or organizational problem; we don’t want them fixed without human investigation.

```
Block12: cvs ntp foo lynx wget \
    serial_console bar sudo mirror_rootvg
foo:
    apt-get -y install foo=0.17-9
    touch $@
bar:
    apt-get -y install bar=1.0.2-1
    echo apple pear > /etc/bar.conf
    touch $@
    ...

```

Listing 1: ISconf makefile package ordering example.

ISconf Version 3

ISconf version 3 was a rewrite in Perl, by Luke Kanies. This version adds more “lessons learned,” including more fine-grained control of actions as applied to target classes and hosts. There are more layers of abstraction between the administrator and the target machines; the tool uses various input files to generate intermediate and final file formats which eventually are fed to ‘make.’

One feature in particular is of special interest for this paper. In ISconf version 2, the administrator still had the potential to inadvertently create unordered change by an innocent makefile edit. While it is possible to avoid this with foreknowledge of the problem, version 3 uses timestamps in an intermediate file to prevent it from being an issue.

The problem which version 3 fixes can be reproduced in version 2 as follows; refer to Listing 1. If both ‘foo’ and ‘bar’ have been executed (installed) on production machines, then the administrator adds ‘baz’ as a prerequisite to ‘bar,’ then this would qualify as “editing prior actions” and create the divergence described in (§A.49).

ISconf version 3, rather than using a human-edited makefile, reads other input files which the administrator maintains, and generates intermediate and final files which include timestamps to detect the problem and correct the ordering.

ISconf Version 4

ISconf version 4, currently in prototype, represents a significant architectural change from versions 1 through 3. If the current feature plan is fully implemented, version 4 will enable cross-organizational collaboration for development and use of ordered change

actions. A core requirement is decentralized development, storage, and distribution of changes. It will enable authentication and signing, encryption, and other security measures. We are likely to replace ‘make’ with our own state engine, continuing the migration begun in version 3. See ISconf.Org for the latest information.

Baseline Management in ISconf

In the ‘Congruence’ section, we discussed the concept of maintaining a fully descriptive baseline for congruent management. In the ‘Describing Disk State’ section, we discussed in general terms how this might be done. In this section, we will show how we do it in isconf.

First, we install the base disk image, usually using vendor-supplied network installation tools. We discuss this process more in [bootstrap]. We might name this initial image ‘Block00’. Then we use the process we mentioned in the ‘ISconf Version 2’ section to apply changes to the machine over the course of its life. Each change we add updates our concept of what is the ‘baseline’ for that class of host.

As we add changes, any new machine we build will need to run isconf longer on first boot, to add all of the accumulated changes to the Block00 image. After about forty minutes’ worth of changes have built up on top of the initial image, it helps to be able to build one more host that way, set the hostname/IP to ‘baseline,’ cut a disk image of it, and declare that new image to be the new baseline. This infrequent snapshot or checkpoint not only reduces the build time of future hosts, but reduces the rebuild time and chance of error in rebuilding existing hosts – we always start new builds from the latest baseline image.

In an isconf makefile, this whole process is reflected as in Listing 2. Note that whether we cut a new image and start the next install from that, or if we just pull an old machine off the shelf with a Block00 image and plug it in, we’ll still end up with a Block20 image with apache and a 2.2.12 kernel, due to the way the makefile prerequisites are chained.

This example shows a simple, linear build of successive identical hosts with no “branching” for different host classes. Classes add slightly more complexity to the makefile. They require a top-level configuration file to define the classes and target them to the right hosts, and they require wrapper script code to read the config file.

There is a little more complexity to deal with things that should only happen at boot, and that can happen when cron runs the code every hour or so. There are examples of all of this in the isconf-2i package available from ISconf.Org.

Cfengine Techniques

Cfengine is likely the most popular purpose-built tool for automated systems administration today. The

cfengine language was optimized for dynamic prerequisite analysis rather than long, deterministic ordered sets.

While the cfengine language wasn't specifically optimized for ordered behavior, it is possible to achieve this with extra work. It should be possible to greatly reduce the amount of effort involved, by using some tool to generate cfengine configuration files from makefile-like (or equivalent) input files. One good starting point might be Tobias Oetiker's *TemplateTree II* [oetiker].

Automatic generation of cfengine configuration files appears to be a near-requirement if the tool is to be used to maintain congruent infrastructures; the class and action-type structures tend to get relatively complex rather fast if congruent ordering, rather than convergence, is the goal.

Other gains might be made from other features of cfengine; we have made progress experimenting with various helper modules, for instance. Another technique that we have put to good use is to implement atomic changes using very small cfengine scripts, each equivalent to an ISconf makefile stanza. These scripts we then drive within a deterministically ordered framework.

In the cfengine version 2 language there are new features, such as the FileExists() evaluated class function, which may reduce the amount of code. So far, based on our experience over the last few years in trial attempts, it appears that a cfengine configuration file that does the same job as an ISconf makefile would still need anywhere from two to three times the number of lines of code. We consider this an open and evolving effort though – check the cfengine.org and Infrastructures.Org websites for the latest information.

Brown/Traugott Turing Equivalence

If it should turn out that the basic logics of a machine designed for the numerical solution of differential equations coincide with the logics of a machine intended to make bills for a department store, I would regard this as the most amazing coincidence that I have ever encountered.

– Howard Aiken, founder of Harvard's Computer Science department and architect of the IBM/Harvard Mark I.

Turing equivalence in host management appears to be a new factor relative to the age of the computing

industry. The downsizing of mainframe installations and distribution of their tasks to midrange and desktop machines by the early 1990's exposed administrative challenges which have taken the better part of a decade for the systems administration community to understand, let alone deal with effectively.

Older computing machinery relied more on dedicated hardware rather than software to perform many administrative tasks. Operating systems were limited in their ability to accept changes on the fly, often requiring recompilation for tasks as simple as adding terminals or changing the time zone. Until recently, the most popular consumer desktop operating system still required a reboot when changing IP address.

In the interests of higher uptime, modern versions of UNIX and Linux have eliminated most of these issues; there is very little software or configuration management that cannot be done with the machine "live." We have evolved to a model that is nearly equivalent to that of a Universal Turing Machine, with all of its benefits and pitfalls. To avoid this equivalence, we would need to go back to shutting operating systems down in order to administer them. Rather than go back, we should seek ways to go further forward; understanding Turing equivalence appears to be a good next step.

This situation may soon become more critical, with the emergence of "soft hardware." These systems use Field-Programmable Gate Arrays to emulate dedicated processor and peripheral hardware. Newer versions of these devices can be reprogrammed, while running, under control of the software hosted on the device itself [xilinx]. This will bring us the ability to modify, for instance, our own CPU, using high-level automated administration tools. Imagine not only accidentally unconfiguring your Ethernet interface, but deleting the circuitry itself...

We have synthesized a thought experiment to demonstrate some of the implications of Turing equivalence in host management, based on our observations over the course of several years. The description we provide here is not as rigorous as the underlying theories, and much of it should be considered as still subject to proof. We do not consider ourselves theorists; it was surprising to find ourselves in this territory. The theories cited here provided inspiration for the thought experiment, but the goal is practical management of UNIX and other machines. We welcome any and all future exploration, pro or con. See the 'Conclusion and Critique' section.

```
# 01 Feb 97 - Block00 is initial disk install from vendor cd,
# with ntp etc. added later
Block00: ntp cvs lynx ...
# 15 Jul 98 - got tired of waiting for additions to Block00 to build,
# cut new baseline image, later add ssh etc.
Block10: Block00 ssh ...
# 17 Jan 99 - new baseline again, later add apache, rebuild kernel, etc.
Block20: Block10 apache kernel-2.2.12 ...
```

Listing 2: Baseline management in an ISconf makefile.

In the following description of this thought experiment, we will develop a model of system administration starting at the level of the Turing machine. We will show how a modern self-administered machine is equivalent to a Turing machine with several tapes, which is in turn equivalent to a single-tape Turing machine. We will construct a Turing machine which is able to update its own program by retrieving new instructions from a network-accessible tape. We will develop the idea of configuration management for this simpler machine model, and show how problems such as circular dependencies and uncertainty about behavior arise naturally from the nature of computation.

We will discuss how this Turing machine relates to a modern general-purpose computer running an automatic administration tool. We will introduce the implications of the self-modifying code which this arrangement allows, and the limitations of inspection and testing in understanding the behavior of this machine. We will discuss how ordering of changes affects this behavior, and how deterministically ordered changes can make its behavior more deterministic.

We will expand beyond single machines into the realm of distributed computing and management of multiple machines, and their associated inspection and testing costs. We will discuss how ordering of changes affects these costs, and how ordered change apparently provides the lowest cost for managing an enterprise infrastructure.

Readers who are interested in applied rather than mathematical or theoretical arguments may want to review the previous sections or skip to the conclusion.

A.1 – A Turing machine (Figure 5) reads bits from an infinite tape, interprets them as data according to a hardwired program and rewrites portions of the tape based on what it finds. It continues this cycle until it reaches a completion state, at which time it halts [turing].

A.2 – Because a Turing machine’s program is hardwired, it is common practice to say that the program *describes* or *is* the machine. A Turing machine’s program is stated in a descriptive language which we will call the *machine language*. Using this language, we describe the actions the machine should take when certain conditions are discovered. We will call each atom of description an *instruction*. An example instruction might say:

If the current machine state is ‘s3’, and the tape cell at the machine’s current head position contains the letter ‘W’, then change to state ‘s7’, overwrite the ‘W’ with a ‘P’, and move the tape one cell to the right.

Each instruction is commonly represented as a quintuple; it contains the letter and current state to be matched, as well as the letter to be written, the tape movement command, and the new state. The instruction we described above would look like:

$s3,W \rightarrow s7,P,r$

Note that a Turing machine’s language is in no way algorithmic; the order of quintuples in a program listing is unimportant; there are no branching, conditional, or loop statements in a Turing machine program.

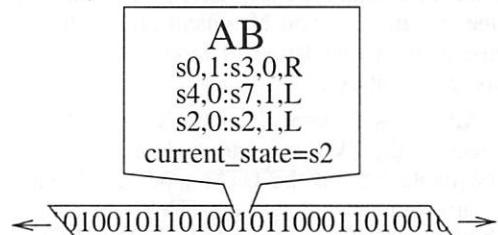


Figure 5: Turing machine block diagram; the machine reads and writes an infinite tape and updates an internal state variable based on a hardwired or stored ruleset.

A.3 – The content of a Turing tape is expressed in a language that we will call the *input language*. A Turing machine’s program is said to either *accept* or *reject* a given input language, if it halts at all. If our Turing machine halts in an accept state, (which might actually be a state named ‘accept’) then we know that our program is able to process the data and produce a valid result – we have validated our input against our machine. If our Turing machine halts because there is no instruction that matches the current combination of state and cell content (§A.2), then we know that our program is unable to process this input, so we reject. If we never halt, then we cannot state a result, so we cannot validate the input or the machine.

A.4 – A Universal Turing Machine (UTM) is able to emulate any arbitrary Turing machine. Think of this as running a Turing “virtual machine” (TVM) on top of a host UTM. A UTM’s machine language program (§A.2) is made up of instructions which are able to read and execute the TVM’s machine language instructions. The TVM’s machine language instructions are the UTM’s input data, written on the input tape of the UTM alongside the TVM’s own input data (Figure 6).

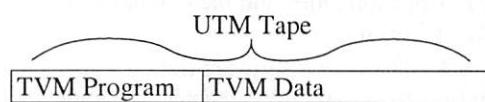


Figure 6: The tape of a Universal Turing Machine (UTM) stores the program and data of a hosted Turing Virtual Machine (TVM).

Any multiple-tape Turing machine can be represented by a single-tape Turing machine, so it is equally valid to think of our Universal Turing Machine as having two tapes; one for TVM program, and the other for TVM data.

A Universal Turing Machine appears to be a useful model for analyzing the theoretical behavior of a “real” general-purpose computer; basic computability theory

seems to indicate that a UTM can solve any problem that a general-purpose computer can solve [church].

A.5 – Further work by John von Neumann and others demonstrated one way that machines could be built which were equivalent in ability to Universal Turing Machines, with the exception of the infinite tape size [vonnewmann]. The von Neumann architecture is considered to be a foundation of modern general purpose computers [godfrey].

A.6 – As in von Neumann's "stored program" architecture, the TVM program and data are both stored as rewritable bits on the UTM tape (§A.4, Figure 6). This arrangement allows the TVM to change the machine language instructions which describe the TVM itself. If it does so, our TVM enjoys the advantages (and the pitfalls) of self-modifying code [nordin].

A.7 – There is no algorithm that a Turing machine can use to determine whether another specific Turing machine will halt for a given tape; this is known as the "halting problem." In other words, Turing machines can contain constructions which are difficult to validate. This is not to say that every machine contains such constructions, but that that an arbitrary machine and tape chosen at random has some chance of containing one.

A.8 – Note that, since a Turing machine is an imaginary construct [turing], our own brain, a pencil, and a piece of paper are (theoretically) sufficient to work through the tape, producing a result if there is one. In other words, we can inspect the code and determine what it would do. There may be tools and algorithms we can use to assist us in this [laitenberger]. We are not guaranteed to reach a result though – in order for us to know that we have a valid machine and valid input, we must halt and reach an accept state. Inspection is generally considered to be a form of testing.

Inspection has a cost (which we will use later):

$$C_{inspect}$$

This cost includes the manual labor required to inspect the code, any machine time required for execution of inspection tools, and the manual labor to examine the tool results.

A.9 – There is no software testing algorithm that is guaranteed to ensure fully reliable program operation across all inputs – there appears to be no theoretical foundation for one [hamlet]. We suspect that some of the reasons for this may be related to the halting problem (§A.7), Gödel's incompleteness theorem [godel], and some classes of computational intractability problems, such as the Traveling Salesman and NP completeness [greenlaw, garey, brooksheat, dewdney].

In practice, we can use multiple test runs to explore the input domain via a parameter study, equivalence partitioning [richardson], cyclomatic complexity analysis [mccabe], pseudo-random input, or other means. Using any or all of these methods, we may be

able to build a confidence level for predictability of a given program. Note that we can never know when testing is complete, and that testing only proves *incorrectness* of a program, not correctness.

Testing cost includes the manual labor required to design the test, any machine time required for execution, and the manual labor needed to examine the test results:

$$C_{test}$$

A.10 – For software testing to be meaningful, we must also ensure code coverage. Code coverage requirements are generally determined through some form of inspection (§A.8), with or without the aid of tools. Coverage information is only valid for a fixed program – even relatively minor code changes can affect code coverage information in unpredictable ways [elbaum]. We must repeat testing (§A.9) for every variation of program code.

To ensure code coverage, testing includes the manual labor required to inspect the code, any machine time required for execution of the coverage tools and tests, and the manual labor needed to examine the test results. Because testing for coverage includes code inspection, we know that testing is more expensive than inspection alone:

$$C_{test} > C_{inspect}$$

A.11 – Once we have found a UTM tape that produces the result we desire, we can make many copies of that tape, and run them through many identical Universal Turing Machines simultaneously. This will produce many simultaneous, identical results. This is not very interesting – what we really want to be able to do is hold the TVM program portion of the tape constant while changing the TVM data portion, then feed those differing tapes through identical machines. The latter arrangement can give us a form of distributed or parallel computing.

A.12 – Altering the tapes (§A.11) presents a problem though. We cannot in advance know whether these altered tapes will provide valid results, or even reach completion. We can exhaustively test the same program with a wide variety of sample inputs, validating each of these. This is fundamentally a time-consuming, pseudo-statistical process, due to the iterative validations normally required. And it is not a complete solution (§A.9).

A.13 – If we for some reason needed to solve slightly different problems with the distributed machines in §A.11, we may decide to use slightly different programs in each machine, rather than add functionality to our original program. But using these unique programs would greatly worsen our testing problem. We would not only need to validate across our range of input data (§A.9), but we would also need to repeat the process for each program variant (§A.10). We know that testing many unique programs will be more expensive than testing one:

$$C_{many} > C_{test}$$

A.14 – It is easy to imagine a Turing Machine that is connected to a network, and which is able to use the net to fetch data from tapes stored remotely, under program control. This is simply a case of a multiple-tape Turing machine, with one or more of the tapes at the other end of a network connection.

A.15 – Building on §A.14, imagine a Turing Virtual Machine (TVM) running on top of a networked Universal Turing Machine (UTM) (§A.4). In this case, we might have three tapes; one for the TVM program, one for the TVM data, and a third for the remote network tape. It is easy to imagine a sequence of TVM operations which involve fetching a small amount of data from the remote tape, and storing it on the local program tape as additional and/or replacement TVM instructions (§A.6). We will name the old TVM instruction set **A**. The set of fetched instructions we will name **B**, and the resulting merger of the two we will name **AB**. Note that some of the instructions in **B** may have replaced some of those in **A** (Figure 7). Before the fetch, our TVM could be described (§A.2) as an **A** machine, after the fetch we have an **AB** machine – the TVM’s basic functionality has changed. It is no longer the same machine.



Figure 7: Instruction set B partially overlays instruction set A, creating set AB.

A.16 – Note that, if any of the instructions in set **B** replace any of those in set **A**, (§A.15), then the order of loading these sets is important. A TVM with the instruction set **AB** will be a different machine than one with set **BA** (Figure 8).



Figure 8: Instruction set BA is created by loading B before A; A partially overlays B this time.

A.17 – It is easy to imagine that the TVM in §A.15 could later execute an instruction from set **B**, which could in turn cause the machine to fetch another set of one or more instructions in a set we will call **C**, resulting in an **ABC** machine:



Figure 9: If instructions from set AB load C, then ABC results.

A.18 – After each fetch described in §A.17, the local program and data tapes will contain bits from (at least) three sources: the new instruction set just copied over the net, any old instructions still on tape, and the

data still on tape from ongoing execution of all previous instructions.

A.19 – The choice of next instruction to be fetched from the remote tape in §A.17 can be calculated by the currently available instructions on the local program tape, based on current tape content (§A.18).

A.20 – The behavior of one or more new instructions fetched in §A.17 can (and usually will) be influenced by other content on the local tapes (§A.18). With careful inspection and testing we can detect some of the ways content will affect instructions, but due to the indeterminate results of software testing (§A.9), we may never know if we found all of them.

A.21 – Let us go back to our three TVM instruction sets, **A**, **B**, and **C** (§A.17). These were loaded over the net and executed using the procedure described in §A.19. Assume we start with blank local program and data tapes. Assume our UTM is hard-wired to fetch set **A** if the local program tape is found to be blank. If we then run the TVM, **A** can collect data over the net and begin processing it. At some point later, **A** can cause set **B** to be loaded. Our local tapes will now contain the TVM data resulting from execution of **A**, and the new TVM machine instructions **AB**. If the TVM later loads **C**, our program tape will contain **ABC**.

A.22 – If the networked UTM machine constructed in §A.21 always starts with the same (blank) local tape content, and the remote tape content does not change, then we can demonstrate that an **A** TVM will always evolve to an **AB**, then an **ABC** machine, before halting and producing a result.

A.23 – Assuming the network-resident data never changes, we can rebuild our networked UTM at any time and restore it to any prior state by clearing the local tapes, resetting the machine state, and restarting execution with the load of **A** (§A.21). The machine will execute and produce the same intermediate and final results as it did before, as in §A.22.

A.24 – If the network-resident data does change, though, we may not be able to rebuild to an identical state. For example, if someone were to alter the network-resident master copy of the **B** instruction set after we last fetched it, then it may no longer produce the same intermediate results and may no longer fetch **C** (§A.19). We might instead halt at **AB**.

A.25 – Without careful (and possibly intractable) inspection (§A.8), we cannot prove in advance whether an **BCA** or **CAB** machine can produce the same result as an **ABC** machine. It is *possible* that these, or other, variations might yield the same result. We can validate the result for a given input (§A.3). We would also need to do iterative testing (§A.12) to demonstrate that multiple inputs would produce the same result. Our cost of testing multiple or partially ordered sequences is greater than that required to test a single sequence:

$$C_{\text{partial}} > C_{\text{test}}$$

A.26 – If the behavior of any instruction from **B** in (§A.22) is in any way dependent on other content found on tape (§A.18, §A.19, §A.20), then we can expect our TVM to behave differently if we load **B** before loading **A** (§A.16). We cannot be certain that a UTM loaded with only a **B** instruction set will accept the input language, or even halt, until after we validate it (§A.3).

A.27 – We might want to rollback from the load or execution of a new instruction set. In order to do this, we would need to return the local program and data tape to a previous content. For example, if machine **A** executes and loads **B**, our instruction set will now be **AB**. We might rollback by replacing our tape with the **A** copy.

A.28 – Due to (§A.26), it is not safe to try to rollback the instruction set of machine **AB** to recreate machine **A** by simply removing the **B** instructions. Some of **B** may have replaced **A**. The **AB** machine, while executing, may have even loaded **C** already (§A.21), in which case you won't end up with **A**, but with **AC**. If the **AB** machine executed for any period of time, it is likely that the input data language now on the data tape is only acceptable to an **AB** machine – an **A** machine might reject it or fail to halt (§A.3). The only safe rollback method seems to be something similar to (§A.27).

A.29 – It is easy to imagine an automatic process which conducts a rollback. For example, in §A.27, machine **AB** itself might have the ability to clear its own tapes, reset the machine state, and restart execution at the beginning of **A**, as in §A.23.

A.30 – But the system described in §A.29 will loop infinitely. Each time **A** executes, it will load **B**, then **AB** will execute and reset the local tapes again. In practice, a human might detect and break this loop; to represent this interaction, we would need to add a fourth tape, representing the user detection and input data.

A.31 – It is easy to imagine an automatic process which emulates a rollback while avoiding loops, without requiring the user input tape in §A.30. For example, instruction set **C** might contain the instructions from **A** that **B** overlaid. In other words, installing **C** will “rollback” **B**. Note that this is not a true rollback; we never return to a tape state that is completely identical to any previous state. Although this is an imperfect solution, it is the best we seem to be able to do without human intervention.

A.32 – The loop in §A.30 will cause our UTM to never reach completion – we will not halt, and cannot validate a result (§A.3). A method such as (§A.31) can prevent a rollback-induced loop, but is not a true rollback – we never return to an earlier tape content. If these, or similar, methods are the only ones available to us, it appears that program-controlled tape changes must be monotonic – we cannot go back to a previous tape content under program control, otherwise we loop.

A.33 – Let us now look at a conventional application program, running as an ordinary user on a correctly configured UNIX host. This program can be

loaded from disk into memory and executed. At no time is the program able to modify the “master” copy of itself on disk. An application program typically executes until it has output its results, at which time it either sleeps or halts. This application is equivalent to a fixed-program Turing machine (§A.1) in the following ways: Both can be validated for a given input (§A.3) to prove that they will produce results in a finite time and that those results are correct. Both can be tested over a range of inputs (§A.9) to build confidence in their reliability. Neither can modify their own executable instructions; in the UNIX machine they are protected by filesystem permissions; in the Turing machine they are hardwired. (We stipulate that there are some ways in which §A.33 and §A.1 are not equivalent – a Turing machine has a theoretically infinite tape, for instance.)

A.34 – We can say that the application program in §A.33 is running on top of an *application virtual machine* (AVM). If the application is written in Java, for example, the AVM consists of the Java Virtual Machine. In Perl, the AVM is the Perl bytecode VM. For C programs, the AVM is the kernel system call interface. Low-level code in shared libraries used by a C program uses the same syscall interface to interact with the hardware – shared libraries are part of the C AVM. A Perl program can load modules – these become part of the program’s AVM. A C or Perl program that uses the `system()` or `exec()` function calls relies on any executables called – these other executables, then, are part of the C or Perl program’s AVM. Any executables called via `exec()` or `system()` in turn may require other executables, shared libraries, or other facilities. Many, if not most, of these components are dependent on one or more configuration files. These components all form an *AVM dependency chain* for any given application. Regardless of the size or shape of this chain, all application programs on a UNIX machine ultimately interact with the hardware and the outside world via the kernel syscall interface.

A.35 – When we perform system administration actions as root on a running UNIX machine, we can use tools found on the local disk to cause the machine to change portions of that same disk. Those changes can include executables, configuration files, and the kernel itself. Changes can include the system administration tools themselves, and changed components and configuration files can influence the fundamental behavior and viability of those same executables in unforeseen ways, as in §A.10, as applied to changes in the AVM chain (§A.34).

A.36 – A self-administered UNIX host runs an automatic systems administration tool (ASAT) periodically and/or at boot. The ASAT is an application program (§A.33), but it runs as root rather than an ordinary user. While executing, the ASAT is able to modify the “master” copy of itself on disk, as well as the kernel, shared libraries, filesystem layout, or any other portion of disk, as in §A.35.

A.37 – The ASAT described in §A.36 is equivalent to a Turing Virtual Machine (§A.4) in the ways described in §A.33. In addition, a self-administered host running an ASAT is similar to a Universal Turing Machine in that the ASAT can modify its own program code (§A.6).

A.38 – A self-administered UNIX host connected to a network is equivalent to a network-connected Universal Turing Machine (§A.14) in the following ways: The host's ASAT (§A.36) can fetch and execute an arbitrary new program as in §A.15. The fetched program can fetch and execute another as in §A.17. Intermediate results can control which program is fetched next, as in §A.19. The behavior of each fetched program can be influenced by the results of previous programs, as in §A.20.

A.39 – When we do administration via automated means (§A.36), we rely on the executable portions of disk, controlled by their configuration files, to rewrite those same executables and configuration files (§A.35). Like the Universal Turing Machine in §A.32, changes made under program control must be assumed to be monotonic; non-reversible short of “resetting the tape state” by reformatting the disk.

A.40 – An ASAT (§A.36) runs in the context of the host kernel and configuration files, and depends either directly or indirectly on other executables and shared libraries on the host's disk (§A.26).

The circular dependency of the ASAT AVM dependency tree (§A.34) forces us to assume that, even though we may not ever change the ASAT code itself, we can unintentionally change its behavior if we change other components of the operating system. This is similar to the indeterminacy described in §A.20.

It is not enough for an ASAT designer to statically link the ASAT binary and carefully design it for minimum dependencies. Other executables, their shared libraries, scripts, and configuration files might be required by ASAT configuration files written by a system administrator – the tool's end user.

When designing tools we cannot know whether the system administrator is aware of the AVM dependency tree (we certainly can't expect them to have read this paper). We must assume that there will be circular dependencies, and we must assume that the tool designer will never know what these dependencies are. The tool must support some means of dealing with them by default. We've found over the last several years that a default paradigm of deterministic ordering will do this.

A.41 – We cannot always keep all hosts identical; a more practical method, for instance, is to set up classes of machines, such as “workstation” and “mail server,” and keep the code within a class identical. This reduces the amount of coverage testing required (§A.10). This testing is similar to that described in §A.13.

A.42 – The question of whether a particular piece of software is of sufficient quality for the job remains intractable (§A.9).

But in practice, in a mission-critical environment, we still want to try to find most defects before our users do. The only accurate way to do this is to duplicate both program and input data, and validate the combination (§A.3). In order for this validation to be useful, the input data would need to be an exact copy of real-world, production data, as would the program code. Since we want to be able to not only validate known real-world inputs but also test some possible future inputs (§A.9), we expect to modify and disrupt the data itself.

We cannot do this in production. Application developers and QA engineers tend to use test environments to do this work. It appears to us that systems administrators should have the same sort of test facilities available for testing infrastructure changes, and should make good use of them.

A.43 – Because the ASAT (§A.36) is itself a complex, critical application program, it needs to be tested using the procedure in §A.42. Because the ASAT can affect the operation of the UNIX kernel and all subsidiary processes, this testing usually will conflict with ordinary application testing. Because the ASAT needs to be tested against every class of host (§A.41) to be used in production, this usually requires a different mix of hosts than that required for testing an ordinary application.

A.44 – The considerations in §A.43 dictate a need for an *infrastructure test environment* for testing automated systems administration tools and techniques. This environment needs to be separate from production, and needs to be as identical as possible in terms of user data and host class mix.

A.45 – Changes made to hosts in the test environment (§A.44), once tested (§A.12), need to be transferred to their production counterpart hosts. When doing so, the ordering precautions in §A.26 need to be observed. Over the last several years, we have found that if you observe these precautions, then you will see the benefits of repeatable results as shown in §A.22. In other words, if you always make the same changes first in test, then production, and you always make those changes in the same order on each host, then changes that worked in test will work in production.

A.46 – Because an ASAT (§A.36) installed on many machines must be able to be updated without manual intervention, it is our standard practice to always have the tool update itself as well as its own configuration files and scripts. This allows the entire system state to progress through deterministic and repeatable phases, with the tool, its configuration files, and other possibly dependent components kept in sync with each other.

By having the ASAT update itself, we know that we are purposely adding another circular dependency beyond that mentioned in §A.40. This adds to the urgency of the need for ordering constraints (§A.45).

We suspect control loop theory applies here; this circular dependency creates a potential feedback loop. We need to “break the loop” and prevent runaway behavior such as oscillation (replacing the same file over and over) or loop lockup (breaking the tool so that it cannot do anything anymore). Deterministically ordered changes seem to do the trick, acting as an effective damper.

We stipulate that this is not standard practice for all ASAT users. But all tools must be updated at some point; there are always new features or bug fixes which need to be addressed. If the tool cannot support a clean and predictable update of its own code, then these very critical updates must be done “out of band.” This defeats the purpose of using an ASAT, and ruins any chance of reproducible change in an enterprise infrastructure.

A.47 – Due to §A.25, if we allow the order of changes to be A, B, C on some hosts, and A, C, B on others, then we must test both versions of the resulting hosts (§A.13). We may have inadvertently created two host classes (§A.41); due to the risk of unforeseen interactions we must also test both versions of hosts for all *future* changes as well, regardless of ordering of those future changes. The hosts may have diverged (see the ‘Divergence’ section).

A.48 – It is tempting to ask “Why don’t we just test changes in production, and rollback if they don’t work?” This does not work unless you are able to take the time to restore from tape, as in §A.27. There’s also the user data to consider – if a change has been applied to a production machine, and the machine has run for any length of time, then the data may no longer be compatible with the earlier version of code (§A.28). When using an ASAT in particular, it appears that changes should be assumed to be monotonic (§A.39).

A.49 – It appears that editing, removing, or otherwise altering the master description of prior changes (§A.24) is harmful if those changes have already been deployed to production machines. Editing previously-deployed changes is one cause of divergence. A better method is to always “roll forward” by adding new corrective changes, as in §A.31.

A.50 – It is extremely tempting to try to create a declarative or descriptive language **L** that is able to overcome the ordering restrictions in §A.45 and §A.49. The appeal of this is obvious: “Here are the results I want, go make it so.”

A tool that supports this language would work by sampling subsets of disk content, similar to the way our Turing machine samples individual tape cells (§A.1). The tool would read some instruction set **P**, written in language **L** by the sysadmin. While sampling disk content, the tool would keep track of some internal state **S**, similar to our Turing machine’s state (§A.2). Upon discovering a state and disk sample that matched one of the instructions in **P**, the tool could

then change state, rewrite some part of the disk, and look at some other part of the disk for something else to do. Assuming a constant instruction set **P**, and a fixed virtual machine in which to interpret **P**, this would provide repeatable, validatable results (§A.3).

A.51 – Since the tool in §A.50 is an ASAT (§A.36), influenced by the AVM dependency tree (§A.34), it is equivalent to a Turing Virtual Machine as in §A.37. This means that it is subject to the ordering constraints of §A.45 and §A.47. If the host is networked, then the behavior shown in §A.15 through §A.20 will be evident.

A.52 – Due to §A.51, there appears to be no language, declarative or imperative, that is able to fully describe the desired content of the root-owned, managed portions of a disk while neglecting ordering and history. This is not a language problem: The behavior of the language interpreter or AVM (§A.34) itself is subject to current disk content in unforeseen ways (§A.35).

We stipulate that disk content can be completely described in any language by simply stating the complete contents of the disk. Cloning, discussed in ‘A Prediction,’ is an applied example of this case. This class of change seems to be free of the circular dependencies of an AVM; the new disk image is usually applied when running from an NFS or ramdisk root partition, not while modifying a live machine.

A.53 – A tool constructed as in §A.50 is useful for a very well-defined purpose; when hosts have diverged (§A.47) beyond any ability to keep track of what changes have already been made. At this point, you have two choices; rebuild the hosts from scratch, using a tool that tracks lifetime ordering; or use a convergence tool to gain some control over them. Cfengine is one such tool.

A.54 – It is tempting to ask “Does every change really need to be strictly sequenced? Aren’t some changes orthogonal?” By *orthogonal* we mean that the subsystems affected by the changes are fully independent, non-overlapping, cause no conflict, and have no interaction each other, and therefore are not subject to ordering concerns.

While it is true that some changes will always be orthogonal, we cannot easily prove orthogonality in advance. It might appear that some changes are “obviously unrelated” and therefore not subject to sequencing issues. The problem is, who decides? We stipulate that talent and experience are useful here, for good reason: it turns out that orthogonality decisions are subject to the same pitfalls as software testing.

For example, inspection (§A.8) and testing (§A.9) can help detect changes which are *not* orthogonal. Code coverage information (§A.10) can be used to ensure the validity of the testing itself.

But in the end, none of these provide assurance that any two changes *are* orthogonal, and like other testing, we cannot know when we have tested or

inspected for orthogonality enough. As in our Perl example in the ‘Ordered Thinking’ section, inspection of high-level code alone is not enough either; we cannot assume that the underlying layers are correct.

Due to this lack of assurance, the cost of predicting orthogonality needs to accrue the potential cost of any errors that result from a faulty prediction. This error cost includes lost revenue, labor required for recovery, and loss of goodwill. We may be able to reduce this error cost, but it cannot be zero – a zero cost implies that we never make mistakes when analyzing orthogonality. Because the cost of prediction includes this error cost as well as the cost of testing, we know that prediction of orthogonality is more expensive than either the testing or error cost alone:

$$\begin{aligned} C_{predict} &> C_{error} \\ C_{predict} &> C_{test} \end{aligned}$$

A.55 – As a crude negative proof, let us take a look at what would happen if we were to allow the order of changes to be totally unsequenced on a production host. First, if we were to do this, it is apparent that some sequences would not work at all, and would probably damage the host (§A.26). We would need to have a way of preventing them from executing, probably by using some sort of exclusion list. In order to discover the full list of bad sequences, we would need to test and/or inspect each possible sequence.

This is an *intractable* problem: the number of possible orderings of M changes is $M!$. If each build/test cycle takes an hour, then any number of changes beyond seven or eight becomes impractical – testing all combinations of eight changes would require 4.6 years. In practice, we see change sets much larger than this; the ISconf version 2i makefile for building HACMP clusters, for instance, has sequences as long as 121 operations – that’s $121!/24/365$, or $9.24*10^{196}$ years. It is easier to avoid unsequenced changes.

The cost of testing and inspection required to enable randomized sequencing appears to be greater than the cost of testing a subset of all sequences (§A.25), and greater than the testing, inspection, and accrued error of predicting orthogonality (§A.54):

$$C_{random} > C_{predict} > C_{partial}$$

A.56 – As a self-administering machine changes its disk contents, it may change its ability to change its disk contents. A change directive that works now may not work in the same way on the same machine in the future and vice versa (§A.26). There appears to be a need to constrain the order of change directives in order to obtain predictable behavior.

A.57 – In contrast to §A.52, a language that supports execution of an ordered set of changes appears to satisfy §A.56, and appears to have the ability to fully describe any arbitrary disk content, as in ‘Describing Disk State.’

A.58 – In practice, sysadmins tend to make changes to UNIX hosts as they discover the need for

them; in response to user request, security concern, or bug fix. If the goal is minimum work for maximum reliability, then it would appear that the “ideal” sequence is the one which is first known to work – the sequence in which the changes were created and tested. This sequence carries the least testing cost. It carries a lower risk than a sequence which has been partially tested or not tested at all.

The costs in §A.8, §A.9, §A.25, §A.54, and §A.55 are related to each other as shown in Figure 10. This leads us to these conclusions:

- Validating, inspecting, testing, and deploying a single ordered sequence (C_{test}) appears to be the least-cost host change management technique.
- Adequate testing of partially-ordered sequences ($C_{partial}$) is more expensive.
- Predicting orthogonality between partial sequences ($C_{predict}$) is yet more expensive.
- The testing required to enable random change sequences (C_{random}) is more expensive than any other testing, due to the $N!$ combinatorial explosions involved.

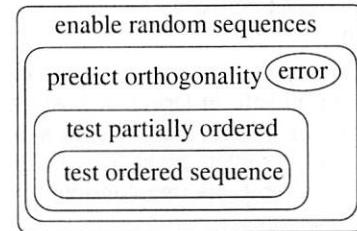


Figure 10: Relationship between costs of various ordering techniques; larger set size means higher cost.

A.59 – The behavioral attributes of a complex host seem to be effectively infinite over all possible inputs, and therefore difficult to fully quantify (§A.9). The disk size is finite, so we can completely describe hosts in terms of disk content, but we *cannot* completely describe hosts in terms of behavior. We can easily test all disk content, but we do not seem to be able to test all possible behavior.

This point has important implications for the design of management tools – behavior seems to be a peripheral issue, while disk content seems to play a more central role. It would seem that tools which test only for behavior will always be convergent at best. Tools which test for disk content have the potential to be congruent, but only if they are able to describe the entire disk state. One way to describe the entire disk is to support an initial disk state description followed by ordered changes, as in ‘Describing Disk State.’

A.60 – There appears to be a general statement we can make about software systems that run “on top of” others in a “virtual machine” or other software-constructed execution environment (§A.34):

If any virtual machine instruction has the ability to alter the virtual machine instruction set, then

different instruction execution orders can produce different instruction sets. Order of execution of these instructions is critical in determining the future instruction set of the machine. Faulty order has the potential to remove the ability for the machine to update the instruction set or to function at all.

This applies to any application, automatic administration tool (§A.36), or shared library code executed as root on a UNIX machine (it also applies to other cases on other operating systems). These all interact with hardware and the outside world via the operating system kernel, and have the ability to change that same kernel as well as higher-level elements of their “virtual machine.” This statement appears to be independent of the language of the virtual machine instruction set (§A.52).

Conclusion and Critique

One interesting result of automated systems administration efforts might be that, like the term ‘computer,’ the term ‘system administrator’ may someday evolve to mean a piece of technology rather than a chained human.

Sometime in the last few years, we began to suspect that deterministic ordering of host changes may be the airfoil of automated systems administration. Many other tool designers make use of algorithms that specifically avoid any ordering constraint; we accepted ordering as an axiom.

With this constraint in place, we have built and maintained many thousands of hosts, in many mission-critical production infrastructures worldwide, with excellent results. These results included high reliability and security, low cost of ownership, rapid deployments and changes, easy turnover, and excellent longevity – after several years, some of our first infrastructures are still running and are actively maintained by people we’ve never met, still using the same toolset. Our attempts to duplicate these results while neglecting ordering have not met these same standards as well as we would like.

In this paper, our first attempt at explaining a theoretical reason why these results might be expected, we have not “proven” the connection between ordering practice and theory in any mathematical sense. We hope we have, however, been able to provide a thought experiment which will help guide future research. Based on this thought experiment, it seems that more in-depth theoretical models may be able to support our practical results.

This work seems to imply that, if hosts are Turing equivalent (with the possible exception of tape size) and if an automated administration tool is Turing equivalent in its use of language, then there may be certain self-referential behaviors which we might want to either avoid or plan for. This in turn would imply

that either order of changes is important, or the host or method of administration needs to be constrained to less than Turing equivalence in order to make order unimportant. The validity of this claim is still an open question. In our deployments we have decided to err on the side of ordering.

On tape size: one addition to our “thought experiment” might be a stipulation that a network-connected host may in fact be fully equivalent to a Universal Turing Machine, including infinite tape size, if the network is the Internet. This is possibly true, due to the fact that the host’s own network interface card will always have a lower bandwidth than the growth rate of the Internet itself – the host cannot ever reach “the end of the tape.” We have not explored the implications or validity of this claim. If true, this claim may be especially interesting in light of the recent trend of package management tools which are able to self-select, download, and install packages from arbitrary servers elsewhere on the Internet.

Synthesizing a theoretical basis for why “order matters” has turned out to be surprisingly difficult. The concepts involve the circular dependency chain mentioned in the section on ‘Ordered Thinking,’ the dependency trees which conventional package management schemes support, as well as the interactions between these and more granular changes, such as patches and configuration file edits. Space and accessibility concerns precluded us from accurately providing rigorous proofs for the points made in the ‘Turing Equivalence’ section. Rather than do so, we have tried to express these points as hypotheses, and have provided some pointers to some of the foundation theories that we believe to be relevant. We encourage others to attempt to refute or support these assertions.

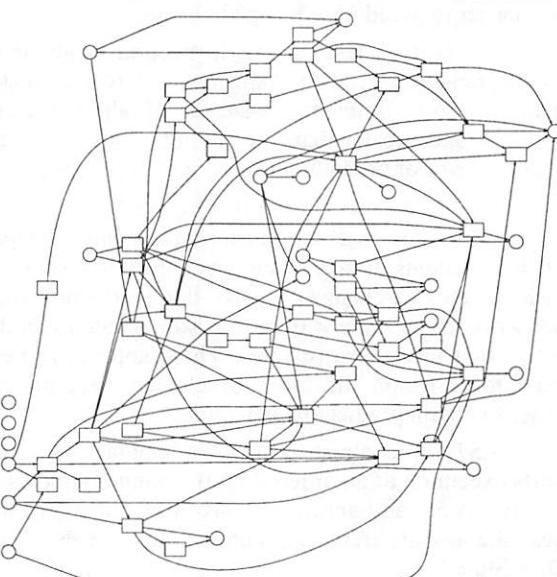


Figure 11: Thread structure of Turing Equivalence assertions.

You are in a maze of twisty little passages, all alike. – Will Crowther’s “Adventure”

There may be useful vulnerabilities or benefits hidden in the structure of the ‘Turing Equivalence’ section. Even after the many months we have spent poring over it, it is still certainly more complex than it needs to be, with many intertwined threads and long chains of assumptions (Figure 11). One reason for this complexity was our desire to avoid forward references within that section; we didn’t want to inadvertently base any point on circular logic. A much more readable text could likely be produced by reworking these threads into a single linear order, though that would likely require adding the forward references back in.

For further theoretical study, we recommend:

- Gödel Numbers
- Gödel’s Incompleteness Theorem
- Chomsky’s Hierarchy
- Diagonalization
- The halting problem
- NP completeness and the Traveling Salesman Problem
- Theory of ordered sets
- Closed-loop control theory

Starting points for most of these can be found in [greenlaw, garey, brookshear, dewdney].

Acknowledgments

We’d like to thank all souls who strive to better your organizations’ computing infrastructures, often against active opposition by your own management. You know that your efforts are not likely to be understood by your own CIO. You do this for the good of the organization and the global economy; you do this in order to improve the quality of life of your constituents, often at the cost of your own health; you do this because you know it is the right thing to do. In this year of security-related tragedies and corporate accounting scandals, you know that if the popular media recognized what’s going on in our IT departments there’d be hell to pay. Still you try to clean up the mess, alone. You are all heroes.

The debate that was the genesis of this paper began in Mark Burgess’ cfengine workshop, LISA 2001.

Alva Couch provided an invaluable sounding board for the theoretical foundations of this paper. Paul Anderson endured the intermediate drafts, providing valuable constructive criticism. Paul’s wife, Jessie, confirmed portability of these principles to other operating systems and provided early encouragement. Jon Stearley provided excellent last-minute review guidance.

Joel Huddleston responded to our recall with his usual deep interest in any brain-exploding problem, the messier the better.

The members of the *infrastructures* list have earned our respect as a group of very smart, very capable individuals. Their reactions to drafts were as good as

rocket fuel. In addition to those mentioned elsewhere, notable mention goes to Dan Hagerty, Ryan Nowakowski, and Kevin Counts, for their last-minute readthrough of final drafts.

Steve’s wife, Joyce Cao Traugott, made this paper possible. Her sense of wonder, analytical interest in solving the problem, and unconditional love let Steve stay immersed far longer than any of us suspected would be necessary. Thank You, Joyce.

About the Authors

Steve Traugott is a consulting Infrastructure Architect, and publishes tools and techniques for automated systems administration. His firm, TerraLuna LLC, is a specialty consulting organization that focuses on enterprise infrastructure architecture. His deployments have ranged from New York trading floors, IBM mainframe UNIX labs, and NASA supercomputers to web farms and growing startups. He can be reached via the Infrastructures.Org, TerraLuna. Com, or stevegt.com web sites.

Lance Brown taught himself Applesoft BASIC in 9th grade by pestering the 11th graders taking Computer Science so much their teacher gave him a complete copy of all the handouts she used for the entire semester. Three weeks later he asked for more. He graduated college with a BA in Computer Science, attended graduate school, and began a career as a software developer and then systems administrator. He has been the lead Unix sysadmin for central servers at the National Institute of Environmental Health Sciences in Research Triangle Park, North Carolina for the last six years. He can be reached at lance@bearcircle.net.

References

- [bootstrap] Traugott, Steve and Joel Huddleston, “Bootstrapping an infrastructure,” *Proceedings of the Twelfth Systems Administration Conference (LISA XII)*, USENIX Association, Berkeley, CA, p. 181, 1998.
- [brookshear] Brookshear, J. Glenn, *Computer Science, An Overview*, (very accessible text), Addison Wesley, ISBN 0-201-35747-X, 2000.
- [centerrun] CenterRun Application Management System, <http://www.centerrun.com>.
- [cfengine] Cfengine, A Configuration Engine, <http://www.cfengine.org/>.
- [church] Church, A., “Review of Turing 1936,” *1937a Journal of Symbolic Logic*, 2, pp. 42-43.
- [couch] Couch, Alva and N. Daniels, “The Maelstrom: Network Service Debugging via ‘Ineffective Procedures’,” *Proceedings of the Fifteenth Systems Administration Conference (LISA XV)*, USENIX Association, Berkeley, CA, p. 63, 2001.
- [cvs] Concurrent Version System, <http://www.cvshome.org>.

- [cvsup] *CVSup Versioned Software Distribution package*, <http://www.openbsd.org/cvsup.html>.
- [debian] *Debian Linux*, <http://www.debian.org>.
- [dewdney] Dewdney, A. K., *The (New) Turing Omnibus – 66 Excursions in Computer Science*, W. H. Freeman and Company, 1993.
- [eika-sandnes] Eika Sandnes, Frode, “Scheduling Partially Ordered Events In A Randomized Framework – Empirical Results And Implications For Automatic Configuration Management,” *Proceedings of the Fifteenth Systems Administration Conference (LISA XV)*, USENIX Association, Berkeley, CA, 2001.
- [elbaum] Elbaum, Sebastian G., David Gable, and Gregg Rothermel, “The Impact of Software Evolution on Code Coverage Information,” *International Conference on Software Engineering*, pp. 170-179, 2001.
- [garey] Garey, Michael R. and David S. Johnson, *Computers and Intractability, A guide to the theory of NP-Completeness*, W. H. Freeman and Company, ISBN 0-7167-1045-5, 2002.
- [godel] Gödel, Kurt, “Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme,” *Monatshefte für Mathematik und Physik*, 38:173-198, 1931.
- [godfrey] Godfrey, M. D. and D. F. Hendry, “The Computer as Von Neumann Planned It,” *IEEE Annals of the History of Computing*, Vol. 15, No. 1, 1993.
- [greenlaw] Greenlaw, Raymond, H. James Hoover, and Morgan Kaufmann, *Fundamentals of the Theory of Computation*, (includes examples in C and UNIX shell, detailed references to seminal works), ISBN 1-55860-474-X, 1998.
- [hagerty] Hagerty, Daniel, hag@ai.mit.edu, personal correspondence, 2002.
- [hamlet] Hamlet, Dick, “Foundations of Software Testing: Dependability Theory,” *Software Engineering Notes, Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, Vol. 19, No. 5, pp. 128-139, 1994.
- [hart] Hart, John and Jeffrey D’Amelia, “An Analysis of RPM Validation Drift,” *Proceedings of the Sixteenth Systems Administration Conference*, USENIX Association, Berkeley, CA, 2002.
- [immunology] Burgess, M., “Computer Immunology,” *Proceedings of the Twelfth Systems Administration Conference (LISA XII)*, USENIX Association, Berkeley, CA, pp. 283, 1998.
- [isconf] *ISconf, Infrastructure Configuration Manager*, <http://www.isconf.org> and <http://www.infrastructures.org>.
- [jiang] Jiang, Tao, Ming Li, and Bala Ravikumar, “Basic Notions in Computational Complexity,” *Algorithms and Theory of Computation Handbook*, p. 24-1, CRC Press, ISBN 0-8493-2649-4, 1999.
- [laitenberger] Laitenberger, Oliver and Jean-Marc DeBaud, “An Encompassing Life Cycle Centric Survey of Software Inspection,” *The Journal of Systems and Software*, Vol. 50, No. 1, pp. 5-31, 2000.
- [lcfg] *LCFG: A Large Scale UNIX Configuration System*, <http://www.lcfg.org>.
- [lisa] *Large Installation Systems Administration Conference*, USENIX Association, Berkeley, CA, <http://www.usenix.org>, 2001.
- [mccabe] McCabe, Thomas J. and Arthur H. Watson, “Software Complexity,” *Crosstalk, Journal of Defense Software Engineering*, Vol. 7, No. 12, pp. 5-9, December 1994.
- [nordin] Nordin, Peter and Wolfgang Banzhaf, “Evolving Turing-Complete Programs for a Register Machine with Self-modifying Code,” *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, Ed. Morgan Kaufmann and L. Eshelman, pp. 318-325, 15-19, ISBN 1-55860-370-0, 1995.
- [oetiker] Oetiker, T., “Template Tree II: The Post-Installation Setup Tool,” *Proceedings of the Fifteenth Systems Administration Conference (LISA XV)*, USENIX Association, Berkeley, CA, p. 179, 2001.
- [opsware] *Opsware Management System*, <http://www.opsware.com>.
- [pikt] “PIKT: ‘Problem Informant/Killer Tool’,” <http://www.piikt.org>.
- [rdist] Cooper, M. A., “Overhauling Rdist for the ‘90s,” *Proceedings of the Sixth Systems Administration Conference (LISA VI)*, USENIX Association, Berkeley, CA, p. 175, 1992.
- [richardson] Richardson, D. J. and L. A. Clarke, “Partition Analysis: A Method Combining Testing and Verification,” *IEEE Trans. Soft. Eng.*, Vol. 11, No. 12, pp. 1477-1490, 1985.
- [rsync] *rsync Incremental File Transfer Utility*, <http://samba.anu.edu.au/rsync>.
- [ssh] *SSH Protocol Suite of Network Connectivity Tools*, <http://www.openssh.org>.
- [sup] Shafer, Steven and Mary Thompson, *The SUP Software Upgrade Protocol*, Carnegie Mellon University, Computer Science Department, 1989.
- [tivoli] *Tivoli Management Framework*, <http://www.tivoli.com>.
- [turing] Turing, Alan M., “On Computable Numbers, with an Application to the Entscheidungsproblem,” *Proceedings of the London Mathematical Society, Series 2*, Vol. 42, pp. 230-265, 1936-37.
- [vonneumann] Von Neumann, John, “First Draft of a Report on the EDVAC,” *IEEE Annals of the History of Computing*, Vol. 15, No. 4, 1993. (From draft produced at the Moore School of Electrical Engineering, University of Pennsylvania, 1945.)
- [xilinx] *Xilinx Virtex-II Platform FPGA*, <http://www.xilinx.com>.

A New Architecture for Managing Enterprise Log Data

Adam Sah – Addamark Technologies, Inc.

ABSTRACT

Server systems invariably write detailed activity logs whose value is widespread, whether measuring marketing campaigns, detecting operational trends or catching fraud or intrusion. Unfortunately, production volumes overwhelm the capacity and manageability of traditional data management systems, such as relational databases. Just loading 1,000,000 records is a big deal today, to say nothing of the billions of records often seen in high-end network security, network operations and web applications. Since the magnitude of the problem is scaling with increases in CPU and networking speeds, it doesn't help to wait for faster systems to catch up.

This paper discusses the issues involving large-scale log management, and describes a new type of data management platform called a Log Management System, which is specifically designed to cost effectively compress, manage and analyze log records in their original, unsummarized form. To quote Tom Lehrer, "I have a modest example here" – in this case commercial software that can store and process logs in parallel across a cluster of Linux-based PCs using a combination of SQL and perl. The paper concludes with some lessons we learned in building the system.

What Is a Log and Why There Is a Problem

Logs are append-only, timestamped records representing some event that occurred in some computer or network device. Once upon a time, logs were used by programmers and system administrators to figure out "what's going on" inside systems, and weren't of much value to business people. That's all changed with the rise of internet-based communication, online shopping, online exchanges, and legal requirements to archive traffic and to protect privacy (a.k.a. avoid getting hacked). Unfortunately, tools to manage log data haven't kept up with the rise in traffic, and people have reverted to building custom tools. This paper describes a general-purpose solution.

As a motivating example, one company we'll call ABC Corp. was using a content delivery network (CDN) to "accelerate" (cache) the results of image requests from their image repository, which stored over 1,000,000 images. Unfortunately, CDNs are expensive and actually slow down the delivery performance for images that aren't frequently accessed. In ABC's application, the access patterns to the images were tied to promotions and other unpredictable criteria. To optimize their use of the CDN, they implemented a log management system (LMS) to capture traffic to the image repository and dynamically choose whether to use the CDN based on the frequency of access. In addition to accelerating their content, the system saved ABC \$10,000 per month in network bandwidth costs.

Broadly, companies like KeyNote, NetRatings (AC Nielsen), DoubleClick, VeriSign, Google and Inktomi provide various hosted internet services, and need to report on their usage (for marketing),

performance (for engineering and 24x7 operations) and conformance to service level agreements (SLAs, also for operations). Network security applications are drowning in log data, coming from system logs, routers, firewalls and intrusion detection systems (IDSs).

There are many reasons that traditional data management solutions cannot effectively manage log data, but the first one that users typically experience is in the sheer volume of log data. For example, here are some online applications and the volumes they generate:

- Loudcloud: over seven GB per day of security-related syslogs
- iPIX: over 20 GB per day for hosting photos on eBay.
- topica: over 60 GB per day logging email traffic.
- TerraLycos: 75 GB per day of weblogs from 12 major web portals.
- shockwave.com: over 24 GB per day of logs about people watching online films and playing online games.
- DoubleClick: over 200 GB/day of records about people seeing online ads.

This paper describes a Log Management System (LMS) which allows network admins to get their arms around their logs without breaking their backs. The author envisions never writing another one-off custom log analyzer, like he had to do for Inktomi (hotbot), bamboo.com (virtual tours) and Internet Pictures (eBay Picture Services).

Previous Solutions and Unresolved Problems

Until recently, most companies discarded operational logs, storing only logs of their financial

transactions. Unfortunately, many companies no longer have this option: you can't figure out why online shoppers are abandoning their shopping carts before completing their purchases unless you look at the page views that **didn't** lead to sales.

One solution people attempt is to sample the data, then run reports against the samples. Similarly, people sometimes run aggregating summaries first, then report against the summaries. Both sampling and summaries suffer from the following issues. First, you have to plan everything in advance – you can't decide later what queries you want, since you've discarded the original data. Buggy sampling/summarization code results in corrupted results forever, another flavor of the “changed your mind” problem. Secondly, sampling is dangerous: if you don't sample across the correct dimension, you get the wrong answer, which can lead to bad business decisions. Lastly, a sample can't tell you whether a something didn't occur. For example, samples and summaries are not useful for security applications or when logs are stored for regulatory reasons.

Another solution is to build an LMS using off-the-shelf components, such as relational databases. Unfortunately, you still have to deal with parsing problems, sequence/session analysis and providing tools for non-experts to use, so the LMS author isn't stuck writing every query. All of these solutions also need to scale up, i.e., they need to parallelize and support paging to disk when running low on RAM. For example, parallel sequence analysis is notoriously tricky. Relational databases solve some of these scaling issues, to a point. Unfortunately, even the fastest databases can't load records as fast as enterprise applications generate them, much less provide the headroom to reload data in case something goes wrong in a load. When it comes time to run queries, they depend on “indexes” (e.g., B-trees) which accelerate some queries and not others, resulting in “cliffs” where performance suddenly degrades for no apparent reason. For example, a regular expression search in a database cannot take advantage of an index. Finally, databases are outrageously expensive, both in hardware, software and people to customize and tune them.

What Does It Mean to Solve the Problem?

Logs are generated, parsed then indexed and compressed – this then allows them to be queried and stored, respectively. As all sysadmins know, management tasks are critical, including reorganizing logs (e.g., for performance) and retiring them when no longer useful. See Figure 1 for a picture.

It is worth noting that it is usually impractical to keep logs “at the edge of the network,” i.e., where they were generated. First, enterprises often require centralized reports, which becomes difficult when logs are separated by slow, unreliable networks and firewalls – or when the log-generating machines lack the

storage or CPU power to effectively answer complex queries. Finally, managing widely distributed systems can be a nightmare, due to heterogeneity of hardware, operating systems, tools, access, etc.

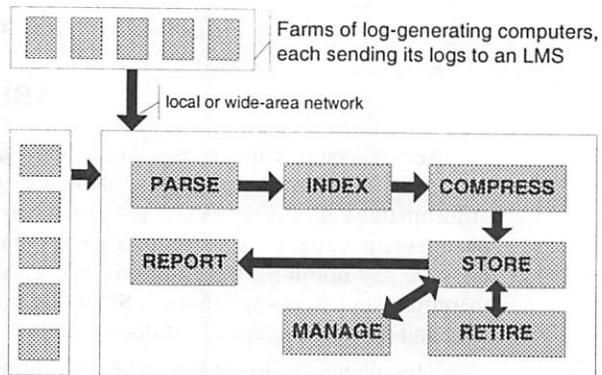


Figure 1: Workflow of log production and analysis.

It is also worth noting that scalability affects everything you do with logs: not only are excellent compressing and indexing basic requirements, but also parallel execution. Intuitively, if you have 5,000 devices generating logs, you probably need more than one collecting the results. Practically speaking, the mainframe-class system capable of keeping up with a large application's traffic costs a ridiculous amount of money.

The Vision

My vision is for a single piece of software to replace the five-minute perl hacks with solid infrastructure for handling log data. In doing so, it is key to create a community which shares scripts to parse various log formats, create various reports, etc. Ideally, there would be a dedicated group of software engineers with the time and talent to invest in features like parallel data management tools, concurrency control so you can load and query data at the same time and connectors to front-end tools like MRTG and CrystalReports.

So we built one. It's in use at places like topica, where they track over one billion emails a month. Running the LMS, five PCs running RedHat 7.1 are able to load more than 20,000 records per second (rps) of weblogs (200-600 bytes/record, depending on the site), then query them at rates of over 250,000 rps. We've handled qmail logs, apache and IIS weblogs, syslogs of various kinds, tuxedo logs and numerous custom logs. Yes, the LMS is a commercial package – it cost us several million dollars to build it.

Design Decisions for a Scalable LMS

Architecturally, the Addamark LMS looks like a webserver, only it listens for requests to a reserved URI (/cgi-app/xmlrpc/execute). If the request contains XML, the server parses the request (including data, e.g., for loading) and returns results, errors and/or progress indicators. Behind the scenes, when you connect to a

server, it parses up your request, and farms it out across the cluster. Each host then parses its piece, matches table and column names against directories and files in its local filesystem (or its NFS-mounted partitions), and processes its chunk of the request.

There is a single config file listing the members of each cluster (`cluster.xml`) and a single config file describing the local config options for the given host (`athttpd.conf`). The local config file, for example, describes the paths to the data, port to listen on, etc. The LMS starts up using an `/etc/init.d` script. Finally, like apache, you can have multiple LMS installations per machine, and as long as they have separate paths, they can run concurrently. In fact, we even conspired to make the lockfiles compatible and the data file (backward) compatible, so two installations can share

the same datastore directories, thereby allowing “rolling upgrades,” which is critical for 24x7 operations, and also critical for avoiding the nightmare of reloading terabytes of data that were loaded over the course of months or years. The diagram in Figure 2 shows what a datastore file-tree might look like. Figure 3 depicts an architecture diagram for the LMS software. As you can see, we tried to avoid reinventing the wheel – even the parallel SQL engine started out as Postgres. As you can see, the network protocol is XML over HTTP, which makes it quite easy to build new clients, including test harnesses.

Loading

Requirements. An LMS should handle any type of logs, not just “standard” ones. Partly, this is

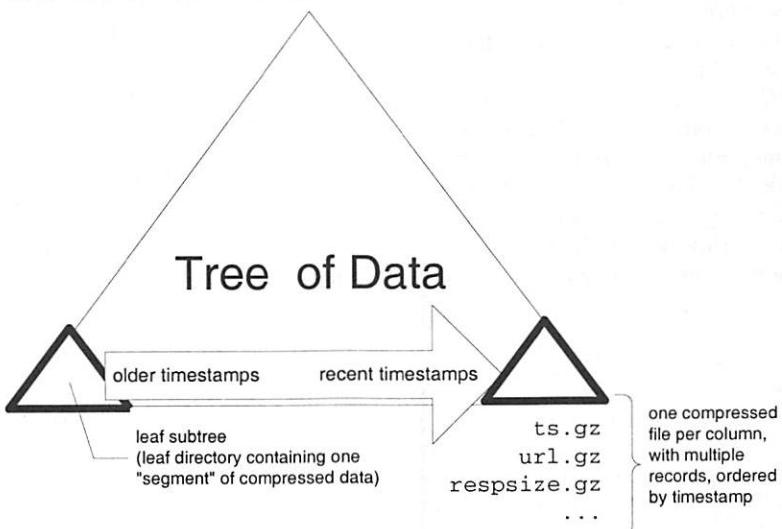


Figure 2: Sample database file tree.

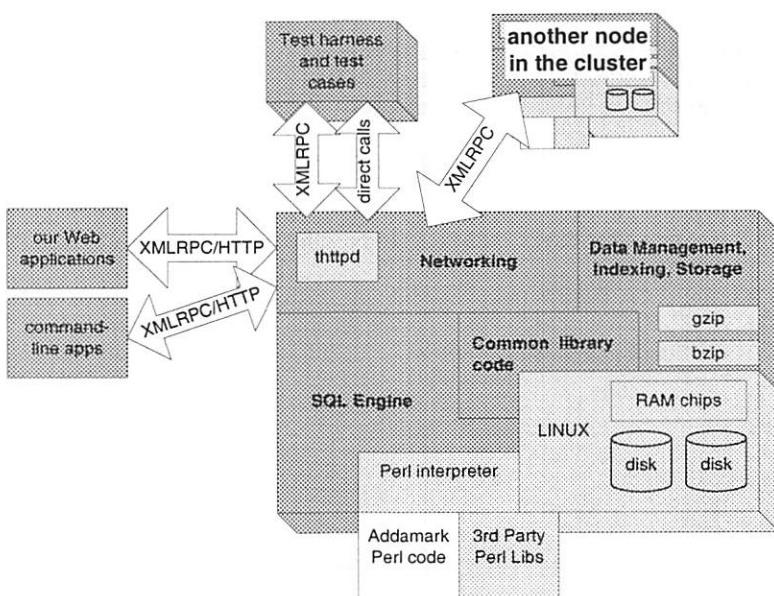


Figure 3: LMS software architecture diagram.

because the apps which most need an LMS are exactly the type who are likely to have large volumes of logs and to customize their log formats to save time or space – or to add special fields they find useful. In our experience, logs tend to exhibit these parsing issues:

- *Quoting and escapification.* Since logs are usually text data separated by some character, you need some way to handle the case when the separator character is present in a given field.
- *Binary data and internationalization.* These days, every data management tool needs to consider these issues.
- *ID fields and reverse DNS.* Logs will often contain fields whose meaning can only be discerned by looking the value up in some other database. An extreme example are IP address fields, where you might want to query on the DNS name they represent. This reverse DNS operation can be very expensive, especially when IP addresses fail to resolve.
- *Variant and XML records, name=value pairs.* Logs with variant records have different “formats” on each line, usually determined by some field in first N fields. This is typically found in custom application logs, rather than logs from commercial devices. However, XML log records are becoming more popular. Sometimes, a field

will contain name=value pairs, e.g., the GET method arguments in a weblog’s URL field.

- *Third party algorithms.* It is sometimes the case that you need (or want) to reuse some third party code to help parse a log record. For example, if one of the fields is encrypted, you almost certainly want to use a third party library for decrypting it.
- *Rejected record handling.* It is an unfortunate reality that log data almost always contains some number of bogus records that fail to parse. It is therefore helpful to have good support for handling rejected records when debugging parsing scripts. Likewise, in cases when “every byte counts” (e.g., legal disputes), you will want to ensure that rejected records aren’t lost.
- *Excluding and double-loading columns.* Sometimes, users will want to discard a column (heresy!), e.g., to save space. Assuming you have excellent data compression, this will be less common than the case when a user will want to “double-load” a column for faster query performance, e.g., load both an IP address as well as its DNS name.

Design Decisions. For performance, we parse logs in parallel across the cluster, using a regular

```
# some example records (for compatibility, we also support hash-comments)
# 199.166.228.8 - - [29/Jan/2002:23:44:37 -0800] "GET / HTTP/1.0" 200 7121
#           "check_http/1.32.2.6 (netsaint-plugins 1.2.9-4)" 0
# 212.35.97.195 - - [29/Jan/2002:23:45:06 -0800] "GET
#           /images/lms_overview_page1.gif HTTP/1.1" 200 17252
#           http://paulboutin.weblogger.com/2002/01/28" "Mozilla/4.0
#           (compatible; MSIE 6.0; Windows NT 5.0; Q312461)" 1
# 62.243.230.170 - - [19/Feb/2002:17:29:43 -0800] "POST /cgi-bin/form.pl
#           HTTP/1.1" 302 5 "http://addamark.com/product/requestform.html"
#           "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)" 1
# vvvvvv this is the regexp used to parse up the records vvvvvvvv
^ . . . . . \[. . . \] " . . . . . " . . . . . " . . . " . . . . $
ClientIP:VARCHAR,unused1:VARCHAR,unused2:VARCHAR,tsStr:VARCHAR,
Method:VARCHAR, Url:VARCHAR, HttpVers:VARCHAR, RespCode:INT32,
RespSize:INT32, Referrer:VARCHAR, UserAgent:VARCHAR, RespTime:VARCHAR
-- ^^^^^^^^^ these are the assigned "parse field" names and datatypes ^^^^^^
-- this is the SQL statement used to transform the parse fields
-- vvvvvvvvvv (from the "stdin" table) into the final table records vvvvvv
--
```

```
SELECT _strptime( tsStr, "%d/%b/%Y:%H:%M:%S %Z") as ts,
ClientIP,
_rev_dns(ClientIP) as ClientDNS, -- perform a reverse DNS lookup
Method,
Url,
HttpVers,
RespCode,
RespSize,
Referrer,
UserAgent,
_int32(RespTime) as RespTime, -- can also parse strings as numbers here,
FROM stdin;
```

Display 1: Example PTL script for loading an NCSA weblog.

expression designed to match single-line records. To handle multi-line records, we pre-process the data before loading, to force records onto one (virtual) line. To provide the flexibility needed, we provide a declarative language based on SQL. Roughly, a load “statement” is a SELECT from a table whose columns are the parse fields, and whose output is used to load the data. To transform a field (e.g., using builtin or third-party functions) simply place a SQL expression in the SELECT statement (the SQL “targets,” as they’re called); to exclude a column, just don’t mention it in the SELECT statement; to double-load a column, mention it twice. For an example, see below (“Load Script Language”).

In addition, we’ve extended our SQL to support functions written in Perl, which you can submit with any SQL statement, either at load- or query-time. In this way, you can write custom parsers and use third party libraries (e.g., perl5 modules) to parse the data. Using the new Inline Perl module (<http://inline.perl.org/>), you can even dynamically load code written in other languages, including C, C++, Java, Ruby, Python, etc. In practice, our users have used the Perl interface in ways we never expected. As an example, one user implemented functions to parse User-Agent tags and look for worms. In this way, he could exclude traffic that wasn’t related to real users, including worms like NIMDA and robot-agents like the google crawler.

Both the regular expression match, SQL statement and any embedded Perl code are all run in parallel across the cluster. In practice, we’ve seen near-linear scaleups because parsing is CPU-intensive once you include all of the “business rules” of real world parsing.

The Addamark parse-transform-and-load “language” (PTL) uses a perl5 regular expression to perform the basic parse, while reusing the SQL and perl engines to perform the transformation. Display 1 shows an example PTL script for loading an NCSA weblog.

Again, it is important to note that the entire PTL script is executed in parallel across the cluster. Thus, even if you embed complex Perl functions or a multitude of complex regular expressions, you’ll still be able to parse tens of thousands of records per second. For example, one customer has a PTL script which calls a home-brewed `parse_useragent` function on every record as it comes in, rather than doing this analysis on every query – although this improves query performance, the real value is in having the table pre-populated with the various browser attributes up-front, which makes query-writing easier.

Putting it together, Figure 4 shows the architecture diagram showing how the LMS loads data; each box represents a thread of control and set of vertically-aligned boxes represents one host. In this example, the cluster is of size three. Typically, a loading client sends the log data to one of the hosts in the cluster, which we call the “master.” Any host can play

“master” for any load request; the job of the master is to break up the datastream into records, and to farm those records out to machines in the cluster.

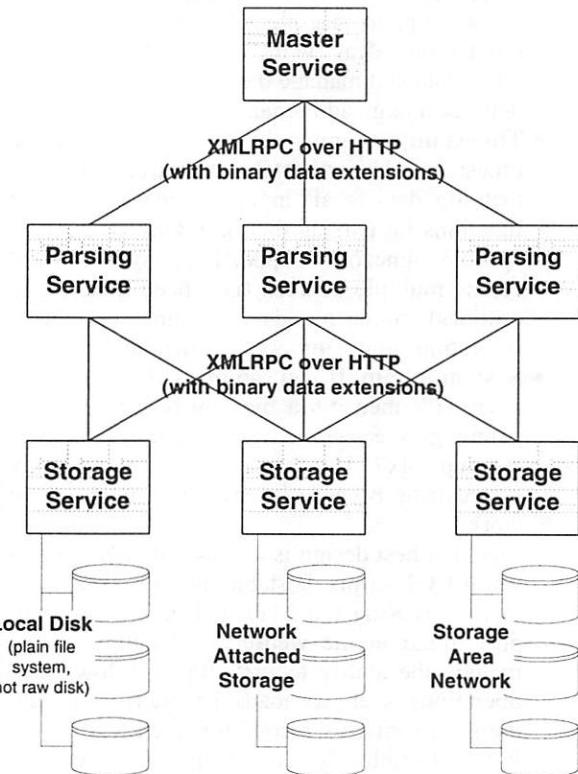


Figure 4: How LMS loads data.

Parsing and storage then happen in parallel, and finally, the records are merged into the existing sets of records on the given storage medium. As suggested by the diagram, the Addamark LMS can store its log data on either local disk, network attached storage (NAS, e.g., NFS), or on a storage area network (SAN, e.g., FiberChannel).

It’s not shown in the diagram, but since the client-server and server-server are identical, it’s straightforward to have client tools load directly into LMS datastore nodes, bypassing the need for a master (and the scalability bottleneck it creates), at the cost of greater configuration complexity.

Storage and Data Management

Requirements. An LMS should automatically handle all indexing, compression, storage, layout and so on – ideally in such a way that queries are then fast to run.

- **Indexing, compression, storage and data management.** Ideally, compression should be as good as GZIP, since managed storage is expensive. Compression ratios should be reasonably stable, as should indexing quality – basic queries should return in the same amount of time regardless of the log data being loaded.

- **data administration tools for giant tables.** Although the requirements are rather specialized, an LMS shares many requirements with databases – you need to control concurrent access, support incremental backup, restore and replication, deal with corrupted datastores, retire data and manage the evolution of the data definition (e.g., add/rename/remove columns).
- **Timestamp support.** Every log record has a timestamp. Unfortunately, it's rarely the case that log data is all in one format (i.e., need functions for parsing them), in GMT (i.e., need flexible timezone support) or synchronized across multiple sources (i.e., need clock sync routines). Some log data contains sub-second resolution, so it's important to handle this.
- **System admin.** If you can make the LMS “not a server,” then it's a big win because system admin gets easier because it doesn't have to stay up 24x7. Unfortunately, this is probably not realistic because it leaves too much of the work for users.
- The next best design is to make the LMS into a set of CGI scripts, hostable inside a webserver, thereby reusing the 24x7 and monitoring support found in the webserver. Example issues include the ability to list/kill/pause/slow LMS operations such as loads or queries, install/uninstall/configure/reconfigure the LMS.
- It is a terrible idea to use threads or custom servers for the LMS, because you then need new tools to manage it and you'll have separate security issues, etc.
- **Security.** Access control, authentication and security are paramount issues in any data management system.

Design Decisions

- **Indexing, compression, storage and data management.** We chose gzip and bzip to perform compression for us, first parsing the data to get the best possible compression ratio. We also employ several tricks that leverage our knowledge of particular types of logs, for example encoding timestamps as delta-offsets from one another, rather than as distinct values. The net result is a compression that almost always beats gzip by a wide margin, sometimes as much as 2x better.
- We chose to store the log data as sets of plain files in the filesystem, one per column. Specifically, the files are laid out as a hierarchical set of directories, broken out by time. For concurrency control, we use lockfiles stored on local disk (e.g., /var/...) because locking over NFS can be flaky, and we figured that some users may want to store their log data on network disks.
- We're careful about touching files, allowing administrators to perform incremental replication,

backup and restore using tools like rsync(1) and find(1). For more information, see “Data Administration” below.

- **Clustering and fault tolerance.** Because the use of multiple computers inherently increases the chances that one system will fail, we included automatic failover. It works by mirroring every record across two hosts in the cluster – each host has a “sibling” for the data it stores. When a computer fails, the hosts that are trying to contact it automatically failover to the sibling, e.g., for running queries. This causes a 50% performance degradation during failures, but 100% performance availability in their absence (unlike RAID). Since perfectly even distribution is not required, when a host fails, loads simply route around both a host and its sibling. Since our clusters typically start at five hosts, this leaves plenty of horsepower even during failures.
- **Timestamp support.** We support TIMESTAMP as a native datatype, represented as a 64-bit integer value of microseconds since the epoch – Jan 1, 1970. The SQL engine has the C library functions strftime() and strptime() built in for parsing and printing TIMESTAMPS, respectively.

Timezone support is offered in all timestamp-related functions, and the SQL engine supports changing its default timezone using the clause “WITH TIMEZONE ...”. To synchronize clocks, load requests from the log-generating devices can include their local clocktime, and the engine will automatically compute the difference and apply it to the log data.

Internally, we store all data in GMT, which is simpler, but which requires users to set the timezone when printing timestamps using “WITH TIMEZONE” or in each formatting call.

- **System admin.** We chose to represent LMS operations as sets of Linux processes, allowing users to use linux tools (e.g., ps(1)) on them. These processes are launched from an off-the-shelf webserver (currently thttpd). In addition to being able to control jobs with per-machine utilities, such as nice(1), we also offer XML-RPC calls to control sets-of-processes across the cluster. We included scripts to perform cluster-wide install/uninstall/reconfigure.
- **Concurrency control.** The Addamark LMS provides a timerange-based concurrency control scheme that enables concurrent updates and queries. For example, retiring data does not block queries or new data loads. Also, two data loads will interleave in such a way as to block neither one, a critical requirement because loads can take a long time, causing timeouts in upstream processes. Unlike generalized database transactions, loads do not perform

reads in between updates, so this interleaving doesn't cause integrity loss.

- **Low-level tools.** Not that this would ever happen, but in practice files can become corrupted through software bugs, disk drive problems, etc. One customer told us a horror story about losing a person-week trying to recover data from a corrupted Microsoft SQL Server database. To reduce this pain, the LMS includes low-level tools to manage the data (e.g., read the files) that does not depend on the LMS being up, and which are resilient to corruption. Likewise, the Addamark LMS includes low-level tools for managing files that are replicated across a cluster, with cluster-wide diff ("cldiff"), synchronization ("clsync"), run-a-command ("clssh"), and so on. These tools are read the same cluster.xml config file, so membership changes affect both the LMS server and the lower-level utilities. However, for obvious reasons, the utilities can override the membership list.
- **Retirement and data evolution.** When the definition of a log changes, e.g., new columns, you need to be able change the definition in the LMS. So-called "schema evolution" can be handled with standard SQL statements such as ALTER TABLE ADD/DROP/RENAME COLUMN. Retiring data works using DELETE FROM, which allows you to define WHERE and DURING clauses to control what data gets deleted. By the time you read this, we'll have implemented a policy manager which provides a nice front-end to handle the most common cases. Also, since the goal of retirement is to save disk space, and since summaries are a tiny fraction of the size of the original data, we're adding facilities (e.g., INSERT INTO SELECT FROM) to retire-to-a-summary and utilities to implement the most common policies. In a clustered system, retiring to offline media can either be done per-system, or unified. The former takes advantage of the file-based storage, while the latter reuses the query mechanism, which already unifies data from across the cluster.
- **Security.** At one level, LMS security is simple: we support SSL access to the cluster. You can also use IP blocking, firewalls and/or VPNs to restrict access to selected clients. In reality, LMS authentication and access control is a complex topic, easily filling a paper all by itself. Simultaneously, this is an area of active development for us, so any information would be obsolete. Look for future reports on the subject.

Querying and Reporting

Requirements. From a high-enough level, querying an LMS is a lot like querying a database. In practice, the workload looks quite different, and the LMS

should be optimized accordingly. First, for some applications, it is important to be able to quickly retrieve the original records – whitespace and all – for example, for legal use. More commonly, users want to get summaries and histograms, such as traffic per unit-time. More sophisticated queries include lookups (e.g., resolve ID fields into live data sources, rather than during loading) and sequencing/sessionizing queries (e.g., recreate web user sessions, or match activity to a given router as an "attack"). In practice, real world use quickly demands custom filters (SQL WHERE clauses), custom counters (SQL aggregates, such as a new type of "SUM") and data sources outside the LMS (virtual/computed tables).

Reporting is the "higher level" functionality around querying, including metadata queries ("what data is available?"), query caching, presentation/formatting (e.g., Microsoft Excel, HTML, XML, etc.) and connectivity (e.g., ODBC, JDBC, DBI/DBD, etc.).

Parallel queries use a similar scheme to loading, but in reverse. Specifically, the SQL DURING and WHERE clauses get executed as part of the filtering service, then the results routed across the cluster to the "compute" services such that every group (i.e., GROUP BY expression) lands on the same host. To support parallel GROUP BY and SLICE BY, the groups are distributed randomly across the cluster. HAVING, which filters groups, is also implemented at the compute layer. Finally, ORDER BY and TOP-n are implemented at the compute layer, and merged together at the master to form the final result. The above description is the general case for simple aggregation queries – fancier cases like JOINs, subqueries, UNIONs, etc. are possible as well, but beyond the scope of this paper, as are the numerous optimizations that we've implemented.

Design Decisions. For querying, we chose to offer a simplified flavor of SQL, make sure it runs in parallel, then use the Perl extension mechanism to handle the custom needs of log applications. To handle sequences/sessions, we extended GROUP BY with SLICE BY, which "slices" a group into multiple groups based on a user-defined predicate. To handle sessions, this predicate can be stateful, e.g., 10 minutes since we've seen activity for a given user. The design of SLICE BY allows the LMS to "sessionize" traffic after it's been loaded – allowing you to change the business definition of a "session" after the fact – and it allows the LMS to sessionize traffic in parallel, a critical requirement (see Figure 5).

We offer "system" tables which contain lists of tables, columns, etc. For caching, formatting and connectivity, we provide a set of client-side tools and connectors. In addition, we opted to use XMLRPC-over-HTTP as our network protocol. This means that you can submit queries to the LMS using curl, lynx or

even a homebrew perl script – without some fancy code library. In practice, partner companies have gotten new clients to work in under an hour, using nothing but examples.

We chose an extended flavor of SQL as the basis for querying the LMS. Display 2 shows the SQL to return the first 100 log records after midnight, Feb 1.

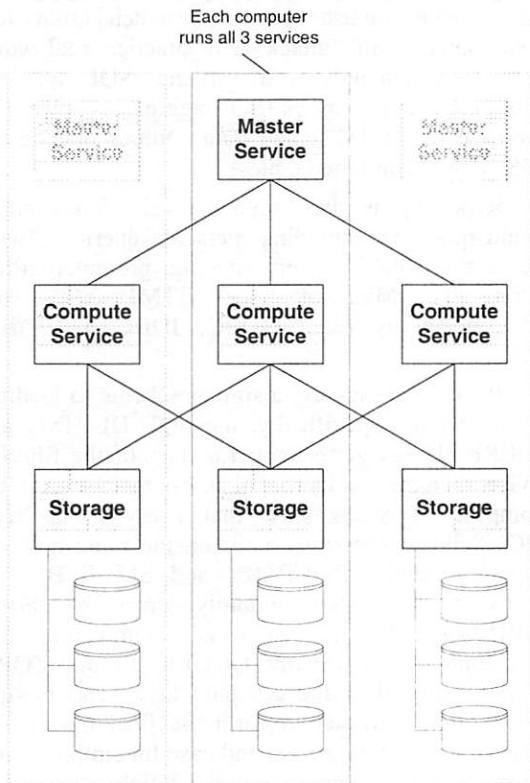


Figure 5: Database querying architecture.

The “WITH” clause sends various parameters to the SQL engine; these can be overridden on the command-line. In this case, we’re telling the engine to produce results in California time, rather than its internal time (GMT).

```
-- dash-dash starts a SQL comment, much like hash (#) in the shell
WITH TIMEZONE 'US/Pacific'
SELECT TOP 100 _timef("%c", ts), hostname, progrname, processID, message
FROM syslog
DURING time('Aug 08 04:00:00 2001'), time('max')
```

Display 2: SQL to return first 100 log records after midnight, Feb 1.

```
WITH TIMEZONE 'US/Pacific'
WITH $end AS _now()
WITH $start AS _timeadd($end, -1, "month")

SELECT _timef("%m/%d/%Y", ts) as 'date'
     , COUNT(*) as 'hits'
FROM example_websrv
GROUP BY 1 -- rollup the hits by date (result column #1)
ORDER BY 1 -- then sort the results by date (result column #2)
DURING $start, $end
```

Display 3: Retrieve a histogram of daily web traffic for previous month.

The SELECT clause tells the engine what columns should appear in the results, and from which table to get them. In this case, we want all of the fields that appear in a (parsed) syslog.

The DURING clause is an Addamark extension which tells the engine which timerange you’re interested in querying, so you don’t accidentally query the whole table. In the rare case when you want to query everything, you can specify “DURING ALL”.

To execute this query, you’d run something like:

```
atquery lms.myco.com:8072 myquery.sql
atquery(1) is our command-line utility for sending your SQL statement to the server, capturing the response (data, errors and/or progress indicators) and pretty-printing it to the screen, file, etc. In this example, “lms.myco.com” would be one of the systems in an LMS cluster. You can even map the LMS hosts into a “virtual IP” behind a load-balancer, which then provides additional fault tolerance.
```

Here’s a more interesting example, retrieving a histogram of website traffic by day for the previous month.

The “WITH \$foo” clauses define expression-macros, which work like C preprocessor macros. We’ve found macros to be lifesavers in practice, especially for clauses like DURING. Even better, the client tools support “include” which includes other files’ worth of macros. This way, you can put the WITH TIMEZONE in a central file, then have every query affected by it. Finally, the tools also support overriding the WITH definitions from the command-line, allowing you to specify the \$start and \$end from the command-line, even though they were also given defaults in the query file. Unlike PL/SQL and other “stored procedure” languages, Addamark SQL uses Perl, i.e., an industry-standard language (Java and C++ coming soon), and the perl code automatically runs in parallel across the cluster of PCs. In practice, the CPUs on modern PCs can execute Perl code amazingly fast, resulting in terrific performance, even for complex algorithms containing numerous regular expressions.

`_now()`, `_timeadd()` and `_timef()` are builtin Addamark functions. We chose to prefix our builtins with underscores to reserve the namespace for other uses. `_now()` returns the timestamp when the query was submitted to the LMS; `_timeadd()` is a builtin function which knows how to add timestamps correctly, including accounting for the timezone. `_timef()` is a function for formatting timestamps as ASCII strings, a direct mapping of the C function `strftime(3)`.

If you also want to return the aggregate bandwidth per day, simply add a third result target – `SUM(respsize)/1024.0/1024.0 AS "MB sent."`

Lastly, to demonstrate the power of embedded Perl, Display 4 shows how to compute the top 25 most popular “pages” in the website. Only, let’s normalize the webpages, so that URLs like `/` and `/index.htm` don’t show up separately.

Lessons Learned

Here are some of the things we learned implementing the LMS:

- **PC clustering changes everything.** Modern networks are very fast and very cheap, and the CPUs on modern PCs are also very fast, so much that it almost always makes sense to trade intra-cluster bandwidth and CPU performance

```
WITH TIMEZONE 'US/Pacific'
WITH $end AS _now()
WITH $start AS _timeadd($end, -1, "month")
-- this defines a new perl function, which can be called from SQL
WITH normalize_url AS 'perl5' FUNCTION <<EOF
sub normalize_url {
my($url) = @_;
# in this site, index.html pages are the same as trailing-slash pages
$url =~ s@/index.s?html?$@/@;
# other rules go here...
#
## uncomment this to send debug messages back to the client tool
## i.e., they're collecting from each of the nodes in the cluster,
## unified and streamed back over the client-connection as out-of-
## band messages.
##
# addamark::dbgPrint("hello, world");
return $url;
}
EOF
SELECT TOP 25 -- returns the first 25 records, assuming there's an
              -- ORDER BY to sort them.
      _perl("normalize_url", url) as url
      , COUNT(*)                      as 'hits'
      , SUM(respsize)/1024.0/1024.0 as 'MB sent'
FROM example_websrv
WHERE respcode < 300    -- ignore HTTP redirects and errors
GROUP BY 1
ORDER BY 2 DESC -- this time, sort by the most-popular-first
DURING $start, $end
```

Display 4: Compute top 25 most popular “pages” in the website.

for other resources, such as RAM capacity or disk I/O. For \$70,000 in hardware, you can put together a system with over 100 GHz of CPU, with more switch bandwidth than the CPUs can saturate, and with 72 terabytes in storage, including a mirror copy.

- **Timestamps are a pain.** It is easy to underestimate the hassles in dealing with timestamps. As a simple example, the default routines for parsing timezones didn’t recognize “PDT” (pacific daylight time) even though it’s produced by the `date(1)` utility. If you don’t solve issues like this, users get annoyed with not being able to cut and paste. Another example is the lack of a “%Z” in `strftime()` so you can capture the timezone from logs which contain per-record timezones. At the user-level, you’ll find logs that are missing critical time fields, such as syslogs that don’t include the year and weblogs lacking the timezone.
- **Buffer the logs.** Originally, we thought that users would want a library for collecting logs – but between syslog, weblogs, etc. users have plenty of logs already, they just need a management system for them! Typically, they “roll” the logs every T time, creating compressed files on the log-generating device. So-called “real-

time analytics” turned out to be a red herring: 99% of applications can live with 5-minute response times because people are involved in the chain, and they can’t react faster than this. Five minutes is plenty to roll a log, compress it and send it to a centralized LMS. Wide-area collection remains a challenge especially across firewalls – but this problem doesn’t seem to have a silver bullet.

- **Tag the data.** The combination of data compression and columnar storage makes “tags” essentially free in most cases (i.e., few unique values). Tagging can be used to provide all sorts of services, and solve all sorts of problems (for example, see “guarantees” below).
- **Guarantees.** In theory, end-to-end atomicity (“once and only once”) across an enterprise requires “two phase commit.” In practice, vendor heterogeneity and the complexity of automated recovery make this impractical. Instead, we rely on store-and-forward (aka buffering) to ensure against data loss. The downside is that duplication becomes possible. Fortunately, the same data-tagging system we use to allow users to track data back to its source also allows the LMS to detect and undo duplicate loads.
- **Packaging matters.** Packaging turned out to be surprisingly important. Our decision to “make the LMS look like apache” was a big win, because it was instantly familiar to users and because the config files were easy to explain. Likewise, replicating all configuration across the cluster made sense to people, while making the LMS resilient to individual machine failures. The biggest win of looking like a webserver, though, was the choice of HTTP as the network protocol, including a complete embedded webserver and a copy of the docs. This meant that our network protocol can be proxied, encrypted, tunneled, etc. – all without special support.

The Future

The existence of a scalable LMS has changed things, but much work remains. First, the combination of fast loading, aggressive data compression and PC disks has all but made log storage “free.” Early users would worry about running out of disk – until they did the math, and realized that even small clusters of PCs could store Years of data. Five PCs alone could store a month of traffic logs from all of Yahoo! This brings us to the second lesson: although you can store years of data online, and access to any (short) timerange is quick, if you want to analyze the whole thing, it’s going to be slow. Therefore, you want to scale the LMS – more CPUs, RAM, etc. – according to the “working set size” rather than disk capacity. Thus, a balanced system would have a tiny disk drive. But extra disk capacity is cheap, so in practice users buy far more than they need. In other words, storage capacity just became free.

Demands from users have suggested our future directions. First, as users build up larger and larger recordsets, they are asking us to provide more and more facilities for managing and reorganizing this data over time. For example, as you grow a cluster, you’ll want to buy the latest, fastest hardware, rather than the same model as when you started. Thus, we’ve recently added a way for the LMS to automatically detect performance differences between machines in the cluster, and load balance between them. Only, unlike with webservers, load balancing parallel SQL requests is quite complex, and is beyond the scope of this paper.

Second, users have started “faking out” the LMS by replicating the files by hand among multiple LMS clusters. While this works to some extent, we can imagine many features that would facilitate distributed, poly-clusters, with (partially) replicated data. Again, this is beyond the scope of this paper.

Thanks

This paper started as a talk at BayLISA April 18, 2002 – thanks to Heather Stern and Strata Rose Chalup for making that happen, and to Marcus Ranum and Rob Kolstad from the mother ship who helped get this paper reviewed and published. Eric Karlson, Nathan Watson, Cimarron Taylor are amazing engineers – they wrote, tested and shipped the LMS v1.0 in a year and a half, when it usually takes 10-15 engineers, and to Christina Noren, who actually made it work in production. Andy Mutz, Steve Levin, Rich Gausell and Katy Ly (iPIX) contributed many design ideas. Thanks to the folks at topica, TerraLycos, AtomShockwave and other customers who waded through early versions. Michael Stonebraker taught me most of what I know about data management systems, and the NOW and Inktomi teams taught me about streaming, pipelining and clustering. We got big help from Mark Soloway, Dave Sharnoff, Sanford Barr, Brent Chapman, Arthur Bohren, Jeff Loomans and Dave Berger. Lastly, thanks to the angel investors who helped pay the bills and keep our spirits up through the dark days of 2001.

Software Availability

The Addamark LMS is a commercial software package available today, with introductory pricing starting around \$75,000 for a complete package. Addamark also offers professional services and support. For more information, please see our website at <http://www.addamark.com/>.

Author Biography

Adam Sah is co-founder and CTO of Addamark Technologies, which makes software to manage enterprise log data, a source of recurring nightmares for him since 1995. Before Addamark, Adam held various management, 24x7 ops and development roles at iPIX (exclusive provider of eBay photohosting, market

leader in virtual tours of real estate), Cohera (distributed database systems, acquired by PeopleSoft) Inktomi (search engines, proxy caches) and Ovid (medical research databases, now a division of Kluwer). Before joining Inktomi as its first employee, Adam was a PhD student at UC Berkeley, where he specialized in distributed databases and programming languages, and invented a way to compile TCL as part of his MS thesis, which Sun added to the language core starting in v8.0. Reach him electronically at asah@addamark.com.

References

- Below are various papers ([LHMWY82], [G94], [HD90]) representing some of the key parallel and distributed database technologies that are used for storing data and processing queries. Also listed are some practitioner reports on using log data that we bumped into along the way ([HDM00], [TH99] and [GB98], [M00] and [M99]). I'm sure there are many I'm neglecting to mention.
- [HDM00] A. Hume, S. Daniels, A. MacLellan. "Gecko: Tracking a Very Large Billing System," *Proc. 2000 USENIX Annual Techn. Conf.*, 2000.
 - [TH99] T. Dunigan, G. Hinkel. "Intrusion Detection and Intrusion Prevention on a Large Network," *Proc. First Workshop on Intrusion Detection and Network Monitoring*, 1999.
 - [GB98] L. Girardin and D. Brodbeck. "A Visual Approach for Monitoring Logs." *Proc. of the 12th Large Installation Systems Administration (LISA) Conf.*, 1998.
 - [HD90] H. Hsiao and D. J. DeWitt. "Chained Declustering: A New Availability Strategy for Multi-processor Database Machines," *Proc. of Sixth Intl. Data Eng. Conf.*, 1990.
 - [LHMWY82] Lindsay, B. G., Haas, L. M., Mohan, C., Wilms, P. F., and Yost, R. A. Computation and Communication in R*: A Distributed Database Manager," *ACM Trans. Comp. Sys.*, 2(1), Feb. 1984.
 - [G94] Goetz Graefe. "Volcano: An Extensible and Parallel Query Evaluation System," *IEEE Trans. on Knowledge and Data Eng.*, 6(1), Feb, 1994.
 - [M00] M. Morton. "Logging and Critical Logs Files: The Decision to Effectively and Proactively Manage System Logging and Log Files," <http://rr.sans.org/securitybasics/logging.php>.
 - [M99] J. Mohr. "Managing Your Log Files," *Linux Magazine*, Nov. 1999, http://www.linux-mag.com/1999-11/guru_04.html.
 - [G02] google search result for 'linux vm "oom killer"', <http://www.google.com/search?hl=en&q=linux+vm+%22oom+killer%22>

MieLog: A Highly Interactive Visual Log Browser Using Information Visualization and Statistical Analysis

Tetsuji Takada & Hideki Koike – University of Electro-Communications

ABSTRACT

System administration has become an increasingly important function, with the fundamental task being the inspection of computer log-files. It is not, however, easy to perform such tasks for two reasons. One is the high recognition load of log contents due to the massive amount of textual data. It is a tedious, time-consuming and often error-prone task to read through them. The other problem is the difficulty in extracting unusual messages from the log. If an administrator does not have the knowledge or experience, he or she cannot readily recognize unusual log messages. To help address these issues, we have developed a highly interactive visual log browser called "MieLog." MieLog uses two techniques for manual log inspection tasks: information visualization and statistical analysis. Information visualization is helpful in reducing the recognition load because it provides an alternative method of interpreting textual information without reading. Statistical analysis enables the extraction of unusual log messages without domain specific knowledge. We will give three examples that illustrate the ability of the MieLog system to isolate unusual messages more easily than before.

Introduction

Administration of computers has become more important than ever because of the increasing role of computers and networks in providing various services to our daily life. It is therefore necessary to conduct them continuously and properly as part of our modern infrastructure.

Computer log inspections are the most fundamental tasks in administration, since most of the events occurring in computers and networks are recorded into log-files. Administrators, therefore, must inspect them periodically. When they find an anomaly in the log-file, they must make an appropriate response as soon as possible. Today's security threats to a computer network increase the importance of log inspections to help detect possible breaches.

Although administrators recognize the importance of log inspection, the task is often not performed regularly at many computer sites. One reason is that it is a tedious and time-consuming task due to the large amount of textual data. Another reason is that it requires skilled knowledge to recognize an unusual message in the log-files.

We have developed a highly interactive log browser, called "MieLog," which uses information visualization and statistical analysis to help alleviate some of the problems involved in log monitoring. The purpose of the system is to assist administrators to inspect computer logs manually. MieLog consists of three main approaches. One is information visualization to improve the recognition load of textual data. Another is a high level of interactivity which makes it easier to filter out or extract information from log data. The last

is statistical analysis which provides inspectors with various tools to help detect unusual messages in the log.

This paper is organized as follows: First, we mention the issues and importance of computer log inspections. The next section presents the system overview and the detail of each module of MieLog. Subsequently, we explain the visualization method of computer logs and interactive functions. Then we show some examples of computer log inspections using MieLog. Finally, we discuss related work and proposed future enhancements.

Problems of Computer Log Inspections

There is no doubt that log inspections are indispensable for computer administration. Administrators, however, regard them as tedious, time-consuming and often unrewarding tasks. Therefore, although some administrators are aware of the importance of the tasks, they hesitate to perform them. Indeed, a recent security survey in Japan shows that such tasks have not been performed sufficiently even by Internet service providers.

We can define log inspections in more specific terms:

1. Administrators retrieve a log and analyze their contents by reading through the messages.
2. Administrators extract unusual messages from the log.

We will consider each of these problems in further detail. The factors that make it difficult to interpret the log contents are that:

- Log messages are recorded as text.
- Logs usually contain a huge amount of data.

- Logs have various kinds of formats and content.

These factors clarify the problems that administrators must face. Administrators must read through the log messages to understand them and they must spend many hours doing so. Thus, it is almost impossible to manually inspect computer logs at a large computer site. Administrators also require specialized knowledge about the logs, since recording formats, contents and existing directories of each log may be completely different. There are many problems with just the log recognition stage in log inspection tasks.

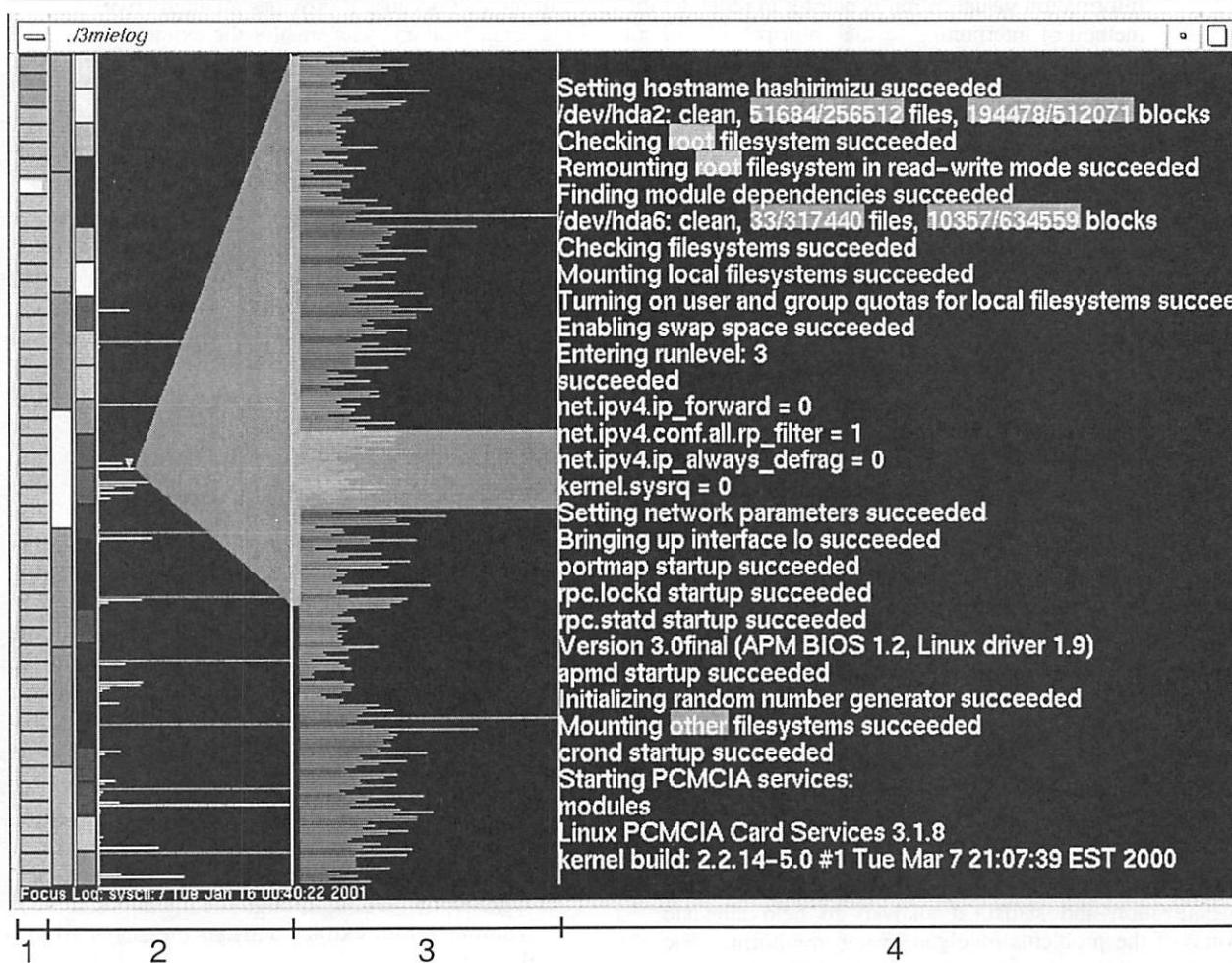
Next, we list the factors that make it difficult to isolate unusual log messages.

- Log messages resulting from a problem or intrusion may be small and buried among other unimportant messages.
- It is difficult to build rules for automatic extraction of all unusual messages.

- There are cases when the administrator cannot determine whether the log message results from an abnormal event or not.

Computer log-files contain various kinds of messages. They contain not only errors and warnings but also operating system status and notice from applications. In general, they contain only a few important messages, while the others are less important messages. Administrators, therefore, must be able to extract the important messages from the log.

The next problem is that it is difficult to formulate the rules for unusual log message extraction. There are two reasons. One is that no administrator knows what constitutes unusual messages for all cases. The other is that the rules are highly dependent on both administrator's knowledge and the environment of the site. We confer that it is important to build rules for extracting known problem log messages. We believe that it is also important to inspect the logs



1. Tag Area 3. Outline Area
2. Time Area 4. Message Area

Figure 1: A display image of MieLog.

periodically to find unknown unusual log messages and rebuild or refine extraction rules based on the new information.

The last problem is that administrators cannot judge the importance of each log message based solely on one kind of log-file. Because each log message contains only partial information about an event that has occurred in a computer, a more reliable judgment requires log messages from multiple log-files. It is therefore necessary to collect other related information from various log-files and analyze them comprehensively. This would require many operations, as well as extensive knowledge and time.

In this paper, we propose the use of information visualization and statistical analysis to address the above problems.

In general, the number of unusual log messages is small in typical log-files. If we obtain frequency information from a log using statistical analysis, it is possible to isolate such log messages. It helps administrators to find truly unusual log messages. Furthermore, MieLog

visualizes frequency information and the log file itself as a figure. It reduces the recognition load of log messages when inspecting them. The reason is that the method of log message recognition changes from "reading" to "looking."

MieLog does not just visualize a log as a figure. It also adds a high level of interactivity. Many of the interactive functions help perform filtering of log messages in various ways. An inspector can execute commands by direct interaction with a visualized figure. The combination of the two features makes it possible to reduce the problems of inspecting logs by humans.

MieLog: System Overview and its Visualization

We developed an interactive log information browser called "MieLog" based on the considerations presented in the previous section. In this section, we describe the system modules of MieLog and the features which address the above mentioned problems.

We used C++ programming language and OpenGL library in development of MieLog. The visual

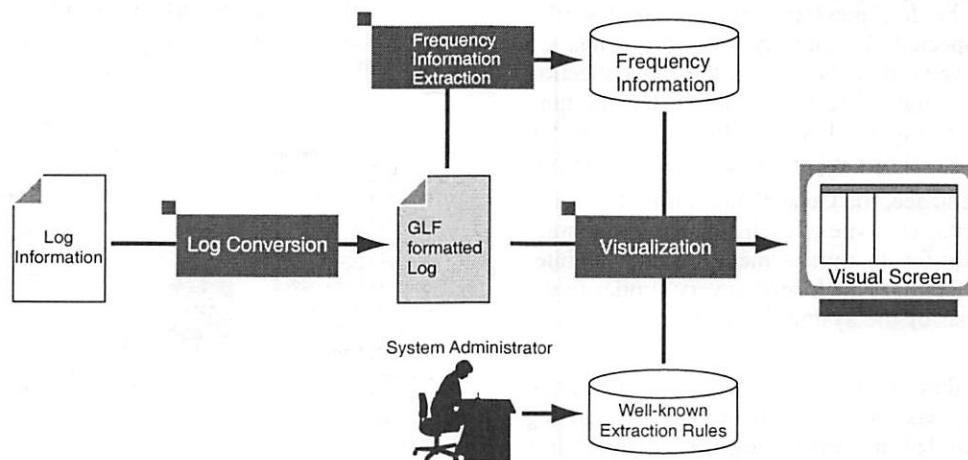


Figure 2: A process module overview of MieLog.

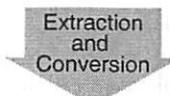
General Log Format (GLF)

Time	Tag 1	Tag 2	Message
------	-------	-------	---------

GLF has the same format at the SYSLOG format, if tag1 is the hostname and tag2 is the program name.

Conversion example to GLF

Jan 10 18:42:02 foo.ac.jp in.telnetd[2424]: connect from someone.else.net



998972366	foo.ac.jp	in.telnetd	connect from someone.else.net
Time	Tag 1	Tag 2	Message

Figure 3: Specification of general log format and conversion example.

screen of MieLog is shown in Figure 1. The screen is composed of four visualization areas. They are the “Tag area,” the “Time area,” the “Outline area” and the “Message area” respectively in order from left to right of the screen. The left three areas visualize different “characteristics” of the log, while the fourth area is for viewing the actual message text. MieLog also visualizes the relationship between each area clearly.

The system modules of MieLog are illustrated in Figure 2. MieLog is composed of three modules. We explain the details of each module in the following sections.

Log Conversion Module

This module converts logs with various recording formats into an intermediate format called the “Generalized Log Format” (GLF).

There are various problems involved in inspecting a log. One is the different recording formats used and the types of contents of each log message. The other is that administrators must have extensive knowledge about the log: where the log-file exists, how to get the log messages and which log-files should be inspected. To simplify these problems, we provide two types of tools. One is the log collection and conversion tool. The other tool is for merging converted logs. Figure 3 shows the syntax of the “Generalized Log Format” and a conversion example.

As you can see, the General Log Format consists of four elements. The type of each element is a character string except for the time element, which is an integer value in seconds. This format is very similar to the message format of the syslog daemon on UNIX systems.

This module contributes to a couple of advantages. MieLog has the ability to browse through log messages recorded in several log-files at one time because the conversion of log message enables the integration of various computer logs into one. The integration of logs are based on the recorded time stamp of each message. These functions reduce the number of operations and time involved when administrators have to inspect several logs. Moreover, this reduces the difficulty of the comprehensive judgment because an inspector receives the time correlation of log messages.

The other role of this module is pre-processing for statistical analysis in order to extract frequency information from the log.

Frequency Information Extraction Module

This module extracts frequency information from GLF formatted log messages using statistical analysis. MieLog uses this information to help extract unusual log messages without pre-defined keywords. This approach is based on the following concept: even if a log has a massive amount of message data, there are generally only a few key messages. In other words,

using such information, we can extract at least the “candidates” of unusual log messages. This assists an inspector in recognizing anomalous messages even if administrators have no prior knowledge or experience about the log.

We explain how to extract frequency information with respect to each element in the GLF as follows:

- **Frequency information regarding the time.**

There are two types of frequency information extracted from the time element of the GLF. One is the number of log messages that occur in each unit of time in a periodical time span. The other is the number of log messages in each unit of time for the entire period of the log.

- **Frequency information regarding the tag.**

This module counts the number of appearances of each tag and keeps them sorted in descending order.

- **Frequency information regarding the message..**

We focus on a word and a phrase in log messages as a unit of the analysis. A phrase in MieLog is defined as a series of two words in a message. The module counts the number of appearances of them (Figure 4).

The diagram illustrates the feature extraction process. It starts with a box labeled "Message information from GLF formatted log". An arrow points from this box to a table titled "Appearance Frequency in Words". Another arrow points from the "Words" table to a table titled "Appearance Frequency in Phrases". A large arrow labeled "Sum up" points from the "Words" table to the "Phrases" table.

Message information from GLF formatted log	
connect from tokyo	
connect from tokyo	
connect from osaka	
connect from kyoto	
connect from osaka	
connect from hakata	
connect from osaka	
connect from osaka	
connect from kyoto	
connect from tokyo	
connect from osaka	
connect from tokyo	
...	

Num. of Appearance	Word
12	connect
12	from
5	osaka
4	tokyo
2	kyoto
1	hakata

Appearance Frequency in Phrases	
Num. of Appearance	Phrase
12	connect from
5	from osaka
4	from tokyo
2	from kyoto
1	from hakata

Figure 4: Feature extraction of messages.

Using MieLog, it is also possible to extract unusual log messages using keywords that an inspector already knows. When an inspector defines keywords, MieLog highlights these words or phrases visually.

Information Visualization Module

MieLog visualizes log messages by combining three kinds of sources: GLF-formatted log messages, frequency information and pre-defined keywords. This module also makes MieLog a highly interactive system. MieLog has a variety of interactive functions to help extract unusual messages. Visualization provides the inspectors with the following two advantages: One is to reduce the load on recognizing a textual message. The other is that it enables administrators to introduce human decision making into judging whether each message seems to be unusual or not.

Visual Representations and Interactive Functions

In this section, we describe the visualization method and interactive functions of MieLog.

Visual Representation of MieLog

The information visualization module creates a visual screen such the one shown in Figure 1. The screen of MieLog is composed of four visual areas as mentioned in the previous section. We will explain the visual representation of each area in this section.

Tag area

The tag area visualizes frequency information of tags as a vertical grid (Figure 5). Each colored tile in the grid represents a corresponding tag information. The number of tiles represents the total number of tags.

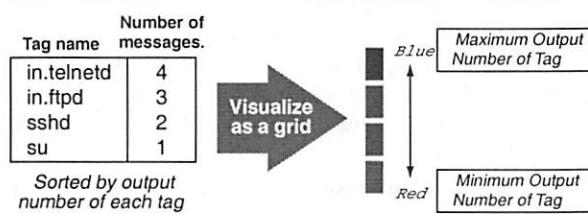


Figure 5: Visualization method of tags in log.

The color of the tiles represents the value of frequency information of each tag. A blue tile indicates that the corresponding tag has the highest frequency value in the log, while a red tile indicates that the corresponding tag has the lowest frequency value. Other tiles with intermediate frequency values have intermediate colors between red and blue. This visualization makes it possible to understand the number and frequency of each tag. The name of each tag is displayed at the bottom of the screen.

MieLog uses another coloring scheme in this area based on the number of tiles, namely the number of tags. The color of each tile is evenly gradated from blue to red. It is easier to distinguish each tile than the colors based on the frequency values.

Time area

The time area is subdivided into three areas. The right-most column of the time area shows a histogram.

The time is assigned from top to bottom, and the value axis is assigned from left to right. It shows how many messages area produced in each unit span. The two left columns in the time area represent the appearance frequency information in different periodic time divisions. As you can see, the left grid has seven tiles and the right has twenty four tiles. This indicates that the left grid represents the appearance frequency information in a week and the right represents them in a day. The representation method of these grids is the same as that of the tag area except for the coloring. The coloring of this area is a gradation between white and black instead of blue and red (Figure 6).

These visual representations make it easier to recognize time-characteristics of the log.

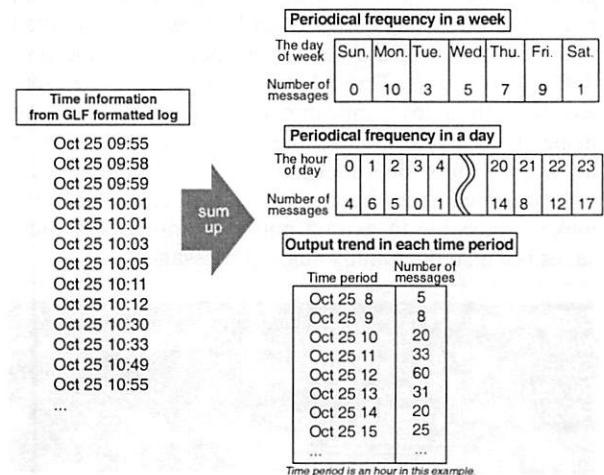


Figure 6: Visualization method of time in logs.

Outline area

This area displays the outline of log messages. Each log message is represented as a colored line. The length of the line is the string length of the log message. The colors of the lines are the same as the grid's colors in the tag area. In other words, the color of each line is assigned to the color of the corresponding tag defined in the tag area (Figure 7).

This visual representation enables administrators to recognize log messages as a visual pattern based on the length and the frequency with which they appear. As a result, it is possible to browse many log messages

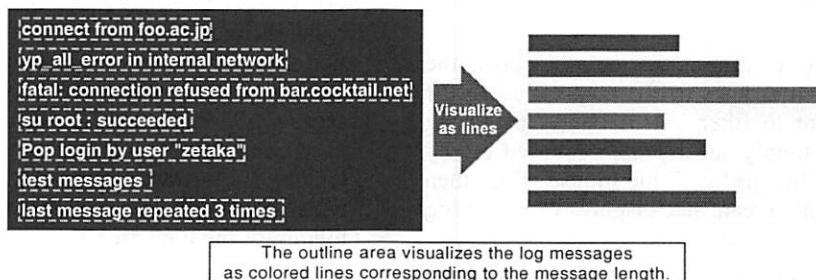


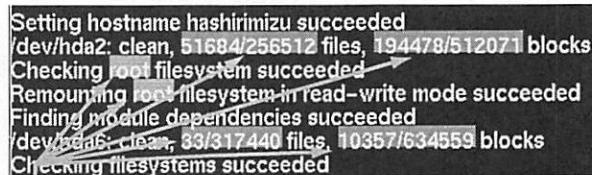
Figure 7: Visualization method of outlines of log messages.

at once, unlike textual representations. Since the line color depends on an appearance frequency of the tag, administrators can also judge whether each message appears to be unusual or not. These provide inspectors with the opportunity to pinpoint unusual log messages before they read the textual log messages.

In the center of this area, a transparent square exists, representing the correlating section between the outline area and the message area. The region of the highlighted square in the outline area is the section displayed in the message area.

Message area

The message area represents actual textual log messages. It is a view similar to that of a text editor, with the exception that it has highlighted words or phrases. Words and phrases highlighted in red and blue (Figure 8). The words and phrases highlighted with red represent the keywords specified in the pre-defined keywords. Those highlighted in blue represent words with a low appearance frequency value. An inspector must define the threshold value manually if he or she wants to extract the words and phrases with a low appearance frequency value. These features make it possible to extract not only known key messages but also potentially suspect messages.



Setting hostname hashirimizu succeeded
 /dev/hda2: clean, 51684/256512 files, 194478/512071 blocks
 Checking root filesystem succeeded
 Remounting root filesystem in read-write mode succeeded
 Finding module dependencies succeeded
 /dev/hda5: clean, 83/317440 files, 10357/634559 blocks
 Checking filesystems succeeded

These words might be valuable for inspectors.

RED highlights pre-defined keywords.

BLUE highlights words which appear with low frequency.

Figure 8: Visualization method of log messages and its features.

Interactive Functions

MiLog has a variety of interactive functions that perform various filtering of log messages. Using these functions, administrators can extract log messages using various visual transformations. This capability effectively assists inspectors by extracting the messages that meet a specific pattern. In this section, we describe the interactive functions of MiLog. We explain them with respect to each visual area.

Tag area

An interactive function in the tag area allows the extraction of log messages with a specific tag. If administrators want to filter log messages using tag information, they simply specify their focused tag by clicking the tile in the grid with the mouse. They then obtain a new visual screen that displays only the log messages with the specified tag.

It is possible for inspectors to specify not only one tag but also multiple tags in the filtering. It is also

possible to specify the tags based on appearance frequency information.

Time area

An interactive function in the time area extracts the log messages based on their recorded time. There are two types of visualization methods in this area: grid visualization and histogram visualization. We explain the interactive function of each of them respectively.

In grid visualization, an inspector can extract log messages based on two types of periodical time spans. They are the hour of the day and the days of the week. The method of filtering, as in the tag area, is to click the tile with the mouse. The administrator can specify multiple tags in a grid. They can also specify multiple tags in two separate grids. In such a case, MiLog extracts the log messages based on the "AND" condition in each time span. In other words, it is possible to extract the log messages that were recorded in 18, 19 and 20 o'clock on Saturday and Sunday just by clicking the five tiles.

In histogram visualization, an inspector can extract the log messages based on the number of log messages in each time span. The filtering method is described as the following. First, the inspectors should select the type of the filtering. There are three filtering conditions: "less than," "nearly equal," and "more than" a threshold value based on the number of log messages in each time span. Next, the inspector should define a threshold value. A vertical line is drawn when the inspector drags the mouse pointer by pushing the right button in the histogram area. That line represents the threshold value for filtering. The inspector can define the threshold value interactively using the visual representation. Finally, the inspector releases the right mouse button to fix the threshold value according to the location of the mouse pointer. The filtering process, then, starts running using the threshold value and the previously defined filtering mode. A visual representation reflects the filtered result.

Outline area

An interactive function in the outline area enables inspectors to extract log messages based on the length of the log message.

Whenever the inspector defines a base length for filtering by manipulating a mouse, they obtain a new visual screen that displays only the messages with a certain length. There are three filtering conditions. The first filtering condition extracts messages shorter than the base length. The second condition extracts messages which are nearly equal to the base length. The third condition extracts messages longer than the base length. The method of filtering is the same as the filtering method based on the output number in the histogram of the time area.

The outline area has another interactive function which enables direct access to the specific log

message for a detailed look, displaying it in the message area. The number of visualized log messages is much greater in the outline area than in the message area. Therefore, many log messages visualized in the outline area are not visible in the message area. If unusual log messages seem to exist in the log messages, it is natural that administrators would want to know the details of them. This interactive function helps them to inspect such log messages more easily.

Message area

An interactive function in the message area extracts log messages that include specific words or phrases. In other words, it is possible to filter log messages by a word or a phrase. The method is described as follows.

First, the inspector should choose words or phrases in the menu. This operation is to decide the unit element for filtering. Second, a filtering condition should be selected. There are three filtering conditions: "and," "or," and "not." These filtering conditions represent the logical relation between selected words or phrases. Filtering with "and" condition extracts the log messages that include all selected words or phrases. Filtering with "or" condition extracts the log messages that include at least one of the selected words or phrases. The above two filtering conditions become effective when an inspector selects more than one word or phrase. Filtering with "not" condition extracts the log messages that do not include the selected words or phrases. It is also possible for an inspector to use this filtering condition when he or she selects only one word or phrase.

The "not" filtering is extremely useful to reduce the amount of visualized messages because it enables the inspector to erase some messages from the inspection target. Using this filtering, the administrators filter out the well-known (i.e., useless) log messages step by step. This function assists the administrator to narrow the inspection target easily and interactively.

There are other kinds of interactive functions that are not closely related to other visual areas.

Defining Keywords and Key Phrases

When the inspector defines keywords or key phrases that are already known as unusual log messages, MieLog highlights them in a manner that the inspector can easily recognize the existence of them. The definition of keywords and key phrases is usually done using another tool such as a text editor. It is, however, possible to define them using MieLog itself through the GUI.

There are two methods of defining keywords or key phrases in MieLog. One is to input them through the GUI. The other method is described as follows. First, the inspector selects a word or a phrase as a keyword or a key phrase on the screen of MieLog using the mouse. Next, he or she starts running the GUI for keyword definition. Then, selected words or phrases

are automatically input in the GUI. If the inspector pushes the "OK" button on the GUI, they are defined as a keywords or key phrases. Namely, the latter method provides a convenient method of input of keywords or key phrases for the inspectors.

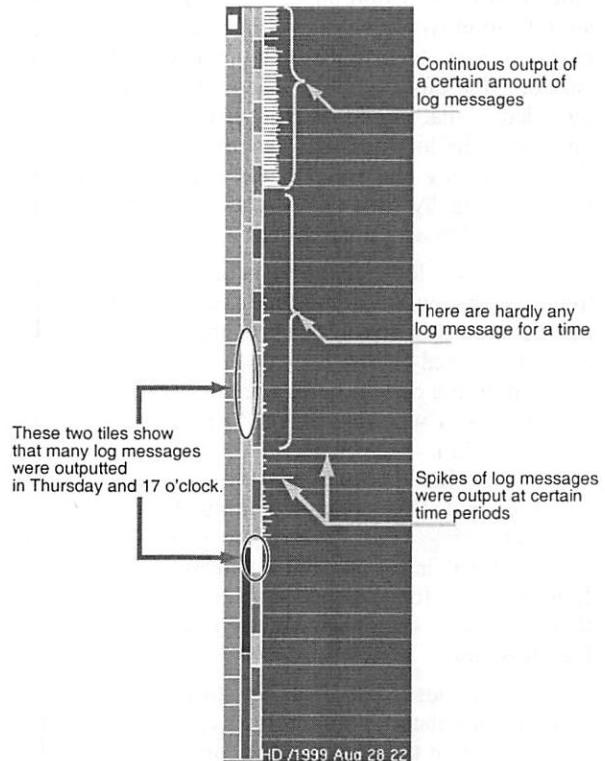


Figure 9: An example of an investigation focusing on time.

Summarizing Log Messages

MieLog has a log summarization function that eliminates the duplication of log messages. Computer log-files generally contain massive amounts of messages. One of the reasons is that relatively unimportant messages are recorded repeatedly in the log. These messages are usually a result of a proper event of the operating system or applications. MieLog, therefore, enables the inspector to summarize log messages interactively in order to reduce the inspection target. If the inspector executes this function, all visualized log messages in MieLog become unique. This makes it possible to reduce the number of log messages and avoid redundant log inspection.

Log Inspection Examples Using MieLog

In this section, we show three examples which demonstrate how to find log messages that seem to show an abnormal behavior using MieLog. We also explain how to use the interactive functions for effective browsing in each case.

An Inspection Example using Log Recording Time Visualization

Figure 9 shows an example of visual representation of the time area.

There are two grids in the time area. Each grid has one bright white tile. This visualization shows that many log messages were recorded at a certain hour of the day and a certain day of the week. In this example, these are 17 o'clock and Thursday. This is a notable indication to help find an unusual log message. Such an indication would be regarded as an abnormal event in many cases. occurrences in many cases. The administrators, therefore, should inspect the log messages recorded in that period of time. It is easy for them to view only the log messages recorded in those periods of time, if they make use of the interactive log message filtering by time. They would simply click the two white tiles with the mouse.

Next, we look at the visualization of an output trend as a histogram in the time area. From the example diagram, it is possible to recognize certain notable activity as listed below:

1. Up until a certain time, a regular number of log messages were continuously recorded in each time span.
2. After this period, no log messages were recorded. This situation continued for a while.
3. Some time later, there are two time spans that recorded large spikes of log messages.

It is possible for the inspectors to recognize all of these indications without actually reading through the log messages.

We propose that there are three specific areas that the administrator should inspect the log in further detail based on the above indications. One is the time when message output was lost. The others are the two time spans when a large number of log messages was recorded.

It is easy to perform these inspections using the interactive function of MieLog. Administrators can easily access the log messages that were recorded in a

specific time span just by clicking the lines in the histogram of the time area. They will get a new visual screen that shows the messages recorded in the specific time span only.

In this example, as a result of the above indications, the inspectors was able to determine the following things: The reason why log messages ceased after a certain time is that a system program did not start after the system configuration was modified. And the massive number of log messages generated in specific time spans were a result of running examination of new software by an another administrator. In this case, both indications did not result from an abnormal event.

We have shown an example in which MieLog was able to represent various indications to the inspector just by focusing on the time area. These indications assisted the inspector to find unusual messages through time trends and frequency, which would have been almost impossible to discover reading through conventional text.

An Inspection Example using Log Outline Visualization

Next, we focus on the visual representation of the outline area. Figure 10 shows three visual representations of the outline area. The left visualization seems to have been made during a normal status. The other visualizations seem to indicate an abnormality.

Focusing on the left visualization, There are three characteristics which can be recognized as follows:

1. Most of the log messages have nearly same length.
2. There are the same series of log messages output towards the middle and the bottom of the visualization.

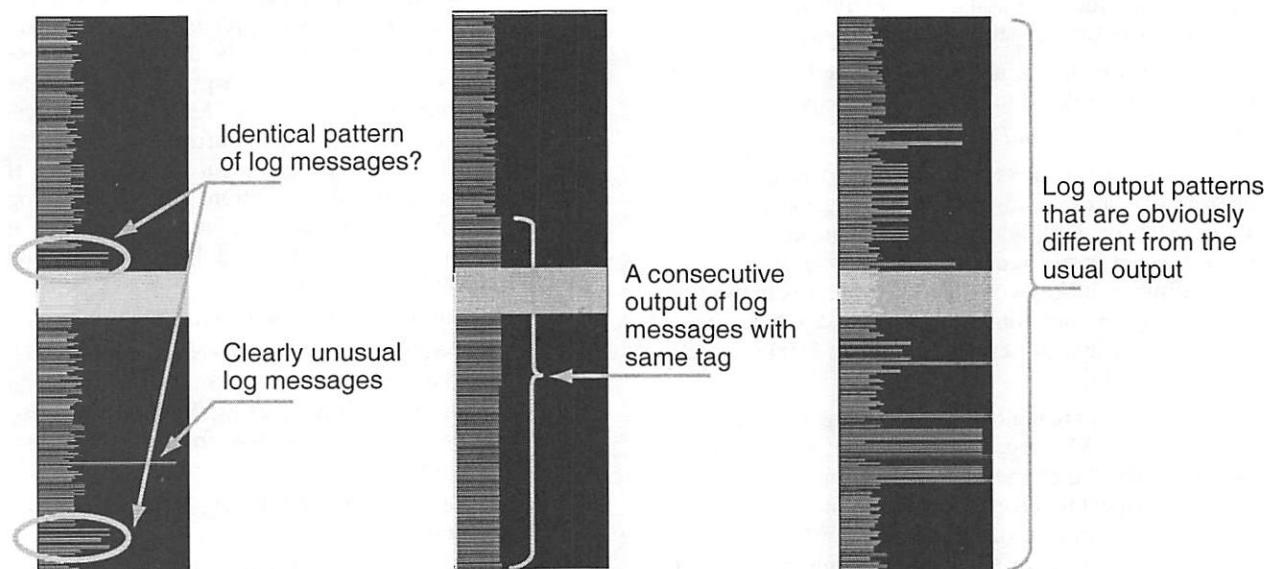


Figure 10: An example of an investigation focusing on log message outlines.

3. There is a message that is clearly longer than the others towards the bottom of the screen.

An administrator should suspect the latter two of the three as an indication of unusual log messages and inspect them in detail. To look at the log messages in detail, the inspector clicks the line in the outline area with the mouse and the text will appear in the message area. If the inspector decides there is something unusual about the messages, he or she needs to investigate the log from the following points of view:

- Were a series of messages being recorded at regular intervals?
- At what time do their messages begin to record?
- Are there any other unusual messages around the time when it started recording their messages?

It is also easy to answer the above questions using interactive functions. The inspector can extract the series of messages using word filtering. The inspector can then easily get the output trend and periodicity of such messages from the time area. He or she, of course, can easily access each message pattern and its surrounding messages. These functions help the inspector to look for unusual message around that time period.

Next, we look at the center image in Figure 10. This figure is clearly different from the log messages outlined in the other two visualizations. The inspector, therefore, easily recognizes an abnormal status at a

glance. The reason is that the log messages in this outline example have various lengths of lines with red color. The inspector knows that these particular log messages rarely appear in this log-file. The inspectors should investigate them in further detail.

We finally focus on the right image in Figure 10. This visualization contains many lines with the same blue color and the same length towards the bottom. This is absolutely unusual status. The line colors are blue and therefore, appear to be a normal status.

We think, however, that the reason why the line colors are blue is because the repeated output of the same message makes its appearance frequency high. The inspector should investigate these messages in further detail.

As seen in these examples, outline visualization enables the inspector to recognize the log messages as a pattern. This feature provides indications to find an unusual message before reading the textual messages. In other words, outline visualization provides another method for extracting unusual log messages other than the frequency information data. The above examples give a glimpse of such ability. This capability greatly depends on using information visualization and introducing a human decision into the judgment.

An Inspection Example Using Log Message Representation with Word and Phrase Highlighting

Finally, we focus on the visual representation of the message area. MieLog represents log messages as text.

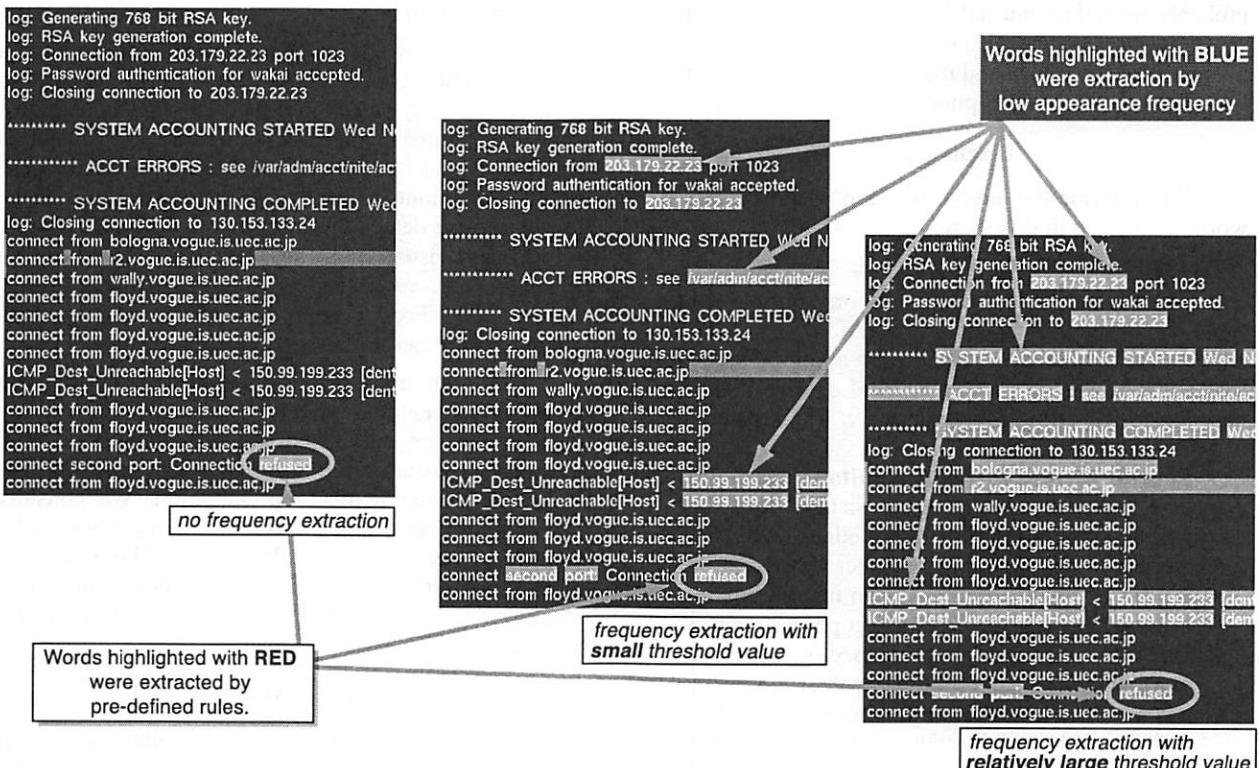


Figure 11: An example of an investigation focusing on log messages.

However, more than just text, MieLog uses highlighting features, which makes it possible to quickly recognize words and phrases with low appearance frequency.

Figure 11 gives three examples of visual representations of the message area. The difference between these visualizations is the method of highlighting the words and phrases for indicating the suspicious log messages. The left visualization is an example using keyword extraction only. There is one word highlighted in red. It means that the word is one of the keywords pre-defined by the inspector.

The other two visualizations are examples using keyword and frequency extraction. There are some words colored blue. It means that they are low appearance frequency words. In the center visualization of Figure 11, the inspector sets the threshold value for low frequency extraction to low. In the right visualization, the inspector sets the threshold value to middle. There are, of course, many more highlighted words with blue color in the right visualization than the center.

If the inspector wants to know the low frequency word, he or she must define the threshold value first. In the current implementation, the inspector must select the threshold value from fixed values using pop-up menu. Then, the inspector clearly recognizes the words and the phrases with low appearance frequency because they are highlighted in blue.

We believe that indicating low appearance frequency words and phrases helps administrators find unusual log messages. Such words and phrases are probably related to unusual log messages. Visual representations reduce the chance of overlooking them in manual log inspections and makes it easier to recognize low frequency words and phrases where they appear.

Discussion

We describe the advantages and proposed future work of MieLog in this section.

Advantages of MieLog

MieLog provides administrators with various tools for inspection using information visualization and statistical analysis. They help administrators to look for unusual log messages. MieLog also makes it possible to inspect logs interactively. The advantages of MieLog are as follows:

- **Logs in various formats simultaneously..** The data analyzed in MieLog consists of logs that were converted into an intermediate format. If there is a log we want to inspect with MieLog, we must first convert it. We can inspect any log using MieLog if the conversion process is provided. It also enables the inspector to inspect more than one log at a time. In other words, we can integrate multiple logs into one, based on the recorded time. It reduces the time and operations involved in inspecting multiple logs. Moreover, such log integration shows the

relationship between the messages that were recorded in separate log-files.

- **Methods which assist in finding unusual log messages.** MieLog extracts appearance frequency information from the log in various graphic visualizations. Their information makes it possible to provide various indications of abnormal events which may be almost impossible to find by administrators reading the textual records. They also allow the inspection of logs from a global point of view. The inspector, for example, can decide whether the log message is unusual or not base on the number of log messages from a specific program. No prior operations and knowledge are needed in this process. Even if the inspector has no knowledge and experiences, he or she makes use of this advantage.
- **Visual Representation and Interactive Functions.** Even if abnormal indications are extracted through statistical analysis, it does not make sense that the inspectors can not recognize them. To resolve this problem, MieLog represents their indications visually in order to recognize them easily and quickly. Using information visualization makes it possible to interact with the visualized information directly. Using interactive functions, the inspector can easily and intuitively extract the log messages that fit a specific condition. We think that it helps to bring the human decision making process to the log inspection task.

We put above advantages in another way.

The greatest advantage of MieLog is to provide a method of anomaly detection for manual log inspection task. An anomaly detection is well-known technique for detecting intrusive behavior in intrusion detection research. There is, however, no system that makes use of such technique. Other log inspection systems make use of misuse detection only, such as keyword search. Information visualization and statistical analysis make it possible to use such technique in inspecting computer log manually. From this point of view, MieLog varies greatly from other log inspection tools.

One more another advantage of MieLog is that it is a human-centered system. MieLog is just a log browser, not an automated log inspection tool. And information visualization makes easier to recognize the log content than textual representation. We consider that there are still a lot of system administrators who want to inspect the log by themselves. There is, however, no system that has functions in order to meet their requests. We believe that MieLog is a tool that has a variety of functions to meet their requests.

Future Works

We describe proposed future enhancements of MieLog in this section. There are two main areas for improvement.

First issue is the log message extraction method. We use the appearance frequency information for extracting unusual log messages. However, not all extracted log messages are unusual log messages. We must evaluate the validation between extracted messages and unusual log messages and refine the extraction method. For example, many numerical values in the log are extracted as low frequency words because they are not recorded repeatedly. Therefore, we should exclude them as targets of statistical analysis.

The next issue is the performance related to the size of logs. The modules that are mainly affected by the size of log are statistical analysis and interactive functions. As the size of log increases, the more time is needed to process the response of the interactive functions. To alleviate this problem, we think it would be better to separate the statistical analysis from the graphical browser. We are also considering using a database.

The last issue is to add a real-time log monitoring feature. This issue has a lot of problems. We think that we must modify the system design largely in order to implement this feature. We also consider that we should be prepare for the another visual expression method for real-time log monitoring. One reason is that MieLog visualizes only a small number of log messages in current representation method. If we use MieLog in real-time log monitoring, MieLog should have an ability to visualize a large number of log messages more than the current because it is easily expected that a lot of log messages are suddenly outputted at a time. If such case occurred, administrators lost the chance to see unusual log messages like above.

In addition, we must prepare log conversion programs for the various type of logs in order to capitalize on the advantages of MieLog. We currently provide only log conversion for UNIX syslog formats.

Related Works

There are a number of log inspection tools already in existence which can be compared with MieLog. In this section, we describe two typical log browsing and inspection systems, and explain how MieLog differs from these types of systems.

One system is "Xlogmaster" [9]. This is a GUI based log monitoring system running on X window system. The main problem of Xlogmaster is that it represents log messages as text and it is harder for inspectors to recognize the log messages. Moreover, it is almost impossible to determine characteristics in the log, such as a message output trend and so on. The inspector must define the keywords in order to extract the unusual log messages. Administrators who have no knowledge and experience of performing log inspection will have difficulty extracting unusual log messages.

The other system is "SeeLog" [1]. SeeLog, like MieLog, represents log messages visually. However, the main problem with SeeLog is that only an outline visualization is available. It is thus difficult to ascertain characteristics of the log. Although SeeLog enables a visual representation of textual log messages, it is difficult for administrators to browse through them. It also has the problem of requiring keyword definitions in order to extract unusual log messages.

There are other log monitoring tools such as Swatch [5], Logsurfer [7] and syslog-*ng* [8]. However, a novice administrator will have difficulty using these tools effectively because the method of unusual log message extraction in these tools is by keyword search.

There are a number of problems with using keyword search as the inherent method of extraction. The first problem is that the inspector must define the keywords. It is, however, difficult for some administrators to do this because not all administrators are aware of the keywords for unusual log message. The second problem is that it is almost impossible to extract unusual log message that are not widely known. The third problem is that extracted log messages are represented as text and will still have a problem with the recognition load of the log messages. The last problem is that these systems do not support log inspection of messages around the suspect message. Administrators must inspect log messages manually around the time of the recorded extracted log message in order to find other related unusual log messages. They also might have to look for another log-files in the same purpose.

Conclusion

In this paper, we have described the interactive visual log browser, named MieLog. MieLog assists human inspection of computer log data. MieLog has three main features which address some problems of log inspection tasks:

1. MieLog reduces the recognition load of the log by using information visualization.
2. MieLog's General Log Format allows the administrator to inspect various kinds of logs at one time.
3. MieLog uses statistical analysis to extract various indications that might closely relate to unusual log messages.

These features provide the following merits. The inspector can inspect more than one log at a time. It is also possible to find an unusual log message even if the inspector has no prior knowledge about them. The most important merit of MieLog is that it brings the human decision making process into the log inspection task.

Future works on MieLog include its evaluation in a practical environment, and the refinement and

extension of message extraction methods and interactive functions.

Availability and Requirements

Regrettably, MieLog is not freely available because we have a plan to be a commercial product. However, we might release a limited version of MieLog in future. The reason is that we have to evaluate it and collect the opinions about MieLog.

Please feel free to contact the author by E-mail to zetaka@computer.org for the current status of MieLog or any related information.

Author Information

Tetsuji Takada is a researcher at the Satellite Venture Business Laboratory in University of Electro-Communications, Tokyo, Japan. His main interests are information visualization and computer security. He received his BS, MS and Dr. of Eng. degrees in information engineering from the University of Electro-Communications in 2000. He is a member of the IEEE. Tetsuji can be reached via email at zetaka@computer.org.

Hideki Koike is an associate professor at the Graduate School of Information Systems, University of Electro-Communications, Tokyo, Japan. His research interests include information visualization, vision-based human-computer interaction, and computer security. He received his BS degree in mechanical engineering from the University of Tokyo in 1986. He received his MS and Dr. of Eng. degrees in information engineering from the University of Tokyo in 1988 and 1991, respectively. He is a member of the IEEE and ACM. Hideki can be reached via email at koike@acm.org.

References

- [1] Eick S. G., M. C. Nelson, and Schmidt J. D.; "Graphical Analysis of Computer Log Files," *Communications of the ACM*, Vol. 37, No. 12, pp. 50-56, 1994.
- [2] Lee, W. and S. Stolfo: "Data Mining Approaches for Intrusion Detection," In *Proceedings of Seventh USENIX Security Symposium*, pp. 79-93, Jan 1998.
- [3] Cox, Kenneth C., Stephen G. Eick, Graham J. Wills, and Ronald J. Brachman, "Visual Data Mining: Recognizing Telephone Calling Fraud," *Journal of Data Mining and Knowledge Discovery*, Vol. 1, No. 2, pp. 225-231, 1997.
- [4] James, A. H., *Audit Log Analysis Using the Visual Audit Browser Toolkit*, Computer Science Department U. C. Davis Technical Report (CSE-95-11), 1995.
- [5] Hansen, S. E. and E. T. Atkins, "Automated System Monitoring and Notification With Swatch," *USENIX Seventh System Administration Conference*, Nov 1993.
- [6] Toshiyuki, Masui, "LensBar – Visualization for Browsing and Filtering Large Lists of Data," In *Proceedings of InfoVis 1998*, pp. 113-120, Oct 1998.
- [7] Ley, W. and U. Ellerman, *Logsurfer*, <http://www.cert.dfn.de/eng/logsurf/>, 1995.
- [8] "BalaBit: The Free Software Company," *syslog-
ng*, <http://www.balabit.hu/products/syslog-ng/>.
- [9] Greve, Georg C. F., *The Xlogmaster*, <http://www.gnu.org/software/xlogmaster/>, 1998.

Process Monitor: Detecting Events That Didn't Happen

Jon Finke – Rensselaer Polytechnic Institute

ABSTRACT

The successful operation of a large scale enterprise information system relies, in part, on the regular and successful completion of many different tasks. Some of these tasks may be fully automated, while others are done manually. One of the challenges we face is detecting when one of these tasks fails (often silently) or is forgotten. While you will eventually learn of these omissions, it is much better to have the system detect them rather than your users! This paper discusses how we implemented a system that watches what we do and reminds us when we (or our computers) forgot to do something.

Introduction

Inspector Gregory: “Is there any other point to which you would wish to draw my attention?”
Holmes: “To the curious incident of the dog in the night-time.”
“The dog did nothing in the night-time.”
“That was the curious incident,” remarked Sherlock Holmes.
From *The Adventure of Silver Blaze* by Arthur Conan Doyle.

At Rensselaer, we manage many of our system and site administration tasks¹ with an Oracle database. For example, we take a data feed from Human Resources to automatically create and expire Unix/email and Windows 2000/Exchange accounts. One aspect of this is that we have many tasks, some run via cron and other scheduling mechanisms, and others run by hand on a regular basis. These tasks generate configuration files [7], [3], web pages (phone directory) [5], process accounting and billing records, update the Active Directory server, and many other things.

One of the problems that we face is knowing when something that is supposed to happen did not. This may be due to a transient file server failure, configuration problems, the failure of a daemon, or simply someone forgetting to do some periodic, yet infrequent task. There are a number of monitoring and logging tools available, from those built into systems such as syslog and programs to help process logs such as *Swatch* [10]. There are also tools that monitor network traffic and system activity such as *Peep* [9] and others. In general, however, all of these systems are looking for things that *are* happening but, like Sherlock Holmes, we are interested in those things that did

¹Originally, I was calling tasks “processes,” but this was causing some confusion on the part of some readers with Unix (or other system) processes. There are still many references to “process” in table and function names, but the intention is to refer to a “task.”

not happen. An earlier project to monitor workstation usage patterns [4] briefly discussed detecting failed workstations by a lack of usage data, but was not pursued. Some other projects to measure system performance via statistical analysis [11, 2] don’t really apply to very low frequency events.

One of the things that I wanted to avoid was writing and maintaining lots of configuration files. Instead, I wanted tasks to report in to a central server when they completed successfully, and then have a nice interface to identify new tasks and quickly set the frequency at which they should reoccur. After that, I don’t want to have to think about that particular task again. Given our heavy use of Oracle in maintaining our system, and that many of the things I was interested in monitoring were already accessing Oracle, an obvious approach for me was to use the database for all of the heavy lifting.

Task Monitor

With that, the Task Monitor project was born. When I use the term “process,” I am not referring to a Unix process, but rather a specific task such as “load printer accounting records,” “update online directory files,” “propagate password changes to Windows,” [8], etc. These may actually be an Oracle job, or part of a job, or a script run out of cron, or even something running on a Windows server.

The information on a task is stored in an Oracle table with the name *Process_Monitor*. The description of this table is broken up into several parts and is included in the appropriate section of the paper. In Figure 1, we have the rough architecture of the Task Monitor system. At the center of things is the *Task_Monitor* package, which acts as an interface between the different tasks and the *Process_Monitor* database table (labeled *Task_Monitor* in the diagram). Tasks communicate via a number of different methods. Some, such as the *Student Upd* package, are running on the oracle server and communicate directly.

Others, such as the Generate_File based modules, connect via SQL*NET. We may also add other interfaces such as syslog or SNMP.

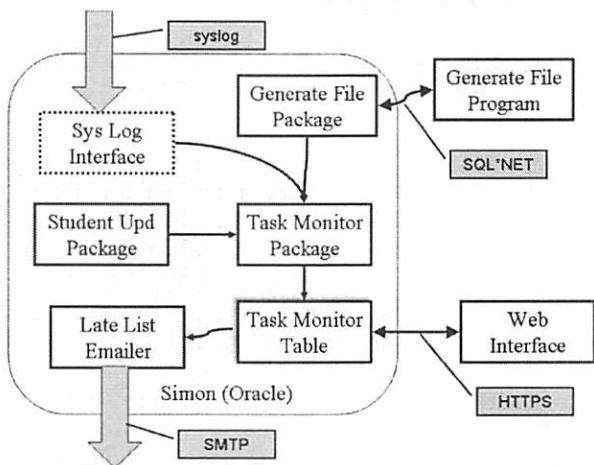


Figure 1: Task monitor architecture.

We also have different ways of getting information out of the Task Monitor system. A program on the

database machine generates email notifications and sends them to interested parties. We also connect via a secure web server for administrative purposes.

Administrative Interface

One of the key parts of this system is the administrative interface, which allows us to set the options for each task. This is implemented via a secure web server.

In Figure two, we have a screen capture of the main web page for the Task Monitor system. This allows you to display different sets of tasks. You select the things you are looking for, and press the "LIST" button. All of the attributes are combined, so the more you select, the more restricted the selection. The first option is to find late or "not late" tasks. Next, you can select the family from the pull down list. There is also a special "NONE" entry that will limit the results to those tasks that are not in a family. You can also select based on those tasks with or without a run delta or schedule, and those tasks that are marked as inactive. Finally, you can restrict the tasks to just those owned by you. (This is run as an administrator; less

Late	<input type="radio"/> Yes <input type="radio"/> No <input checked="" type="radio"/> Null	(Null - All processes)
Family	<select></select>	Blank - All entries
Run Delta	<input type="radio"/> Yes <input type="radio"/> No <input checked="" type="radio"/> Null	(Null - All processes)
Inactive	<input type="radio"/> Yes <input type="radio"/> No <input checked="" type="radio"/> Null	(null - all processes)
My Processes	<input type="radio"/> Yes <input checked="" type="radio"/> No	Just processes owned by me

Figure 2: Main web page.

privileged users will just get their own tasks.) A sample of this list can be seen in Figure seven.

In Figure three, we have a sample web page of the “Logins-Oracle IDs” task. The objective of this task is to create Oracle accounts based on changes in the Logins table. This task is considered part of the “daily run,” a set of activities performed by our User Services staff. From this page, we could move the task to another family using the pull down list, or create a new family by entering the name in the “New Family” box. (This box does not appear if you are not an administrator; you are limited to existing families.) In this case, we don’t care what system this task runs on, only that it is run; so we leave the “System” box empty. The person who normally does this task is Judy Shea, so we have listed her as the owner. If we wanted to let other folks know if this task was late, we could provide a list of email addresses in the “Contact List” box. The next thing we can specify is the “Run Delta,” which is specified as DAYS HOURS:MIN:SEC. In this case, we want this to be run every 28 hours (one day and four hours). This gives Judy a little bit of flexibility in when she does the actual run. The “Notify Delta” is like specified like the “Run Delta” and controls how frequently we report a missed task. Lastly, we can mark as task as inactive, which turns off all notification.

The next part of the page reports on information collected at the last run. The “Next Run” is the time

and date when we next expect this task to be run. If that time was passed, this would be in bold face and be marked as late. (This is only set if there is a “Run Delta” set.) The “Last Run,” which is always available lists the time and date of the most recent run. Currently, the only way to get a task into the system is to run it, so there is always a “Last Run” entry. Next up is when we last notified someone about a late process, and when we expect to send the next notification (assuming a run has not been completed.) There is also space for a free format comment on the task.

When a run is recorded, the system attempts to capture the host OS username and the hostname. There is also an option when recording a task to include a comment; this is task specific. We also record the Oracle user and what Oracle package made the call.

Identifying a Task

A lot of the tasks we are interested in monitoring are site wide. For example, the process that regenerates the directory web pages only needs to run on a single system and write a file into our central file server. There is no need to run it on each of our production web servers, as they all use the same central file server. In other cases, however, we want to be sure that each server is reporting in. An example of this would be the process that collects the printer accounting logs. We want to ensure that each print server is

Logins-Oracle IDs

Name	Logins-Oracle IDs
Family	Daily Run
New Family	
System	
Owner	Judy Shea
Contact List	
Run Delta	1 04:00:00
Notify_Delta	
Inactive	<input type="radio"/> Yes <input checked="" type="radio"/> No
Next Run	18:54:09 17-Jul
Last Run	14:54:09 16-Jul
Last Notify	
Next Notify	
Run User	sheaj
Run Host	vcmr-42
Run Comments	
Oracle User	OPSSHEAJ
Procedure Name	SIMON_USER_MGMT
Comments	Creates Simon userids for RCS users. Should happen when new ac
Update	Delete

Figure 3: Sample task web page.

reporting its activity on a regular basis. It may also be useful to identify the user that is running the process.

For the purposes of this project we identify a process by a process name and the system on which it runs. In this way, if the same process runs on two different machines, it will be considered two different tasks for the purposes of monitoring. This has not been a problem with the site wide tasks, as they are generally run via cron or some other trigger on a single designated machine and by the same user (daemon or equivalent). Since most of the testing and development takes place on a different machine, this isolates the development and test runs from the production runs. We also record the name of the person who ran the task, but this is currently not used to distinguish tasks.

Parenting a Process

In order to help with sorting and grouping, each process can be assigned to a "Family." These are general categories such as "Accounting," "Daily Run," "File Gen," etc. When a new process is entered, it will not have a family assigned to it. This works well, as the administrative web tool can display all tasks in

a family, or those without a family. This provides a quick and easy way to identify new tasks.

When we encounter a new process, we assign it an owner and a family. We can also link it to a service in our ServiceTrak [6] so a the page displaying service information can also include details on some of these tasks. Once a process has an owner, that owner can use the same web tool to finish the setup by assigning a run delta or schedule, or just marking it as inactive.

Reporting a Task

The first challenge of this project was to find ways for tasks to report that they ran. When a process reports in via one of the methods described below, we first see if we have an existing process record. If not, we create a new one; otherwise, we update some of the information and, if there is a run schedule or delta, we then calculate the next run time and save the record. If there were previous error conditions, we clear them as well.

Direct PL/SQL Procedure Call

A number of the tasks that we want to watch are written entirely in PL/SQL and are run on the database

Name	Type	Size	Description
Entry_Id	Number		A unique key to identify this record.
Name	varchar2	32	The name or external identifier for this process.
System_Id	Number		The unique identifier of this system in the hostmaster and service database.
Run_Host	varchar2	128	The hostname where this last ran. Useful when the System_Id can not be determined.
Run_User	varchar2	32	The name of the host system user who ran the process (if available).

Table 1: Process_Monitor table – identification.

Name	Type	Size	Description
Family	varchar2	32	An identifier used to group tasks for display and reporting.
Owner	Number		The internal identifier of the person who "owns" this process.
Service_Id	Number		The internal identifier of the service (see ServiceTrak) that this process supports.
Run_Delta	Number		The maximum allowable time in seconds between the last run and the next run of this process.
Run_Schedule	varchar2	128	A crontab format schedule.
Inactive	varchar2	1	A flag indicating that the current entry is inactive and should be ignored.

Table 2: Process_Monitor table – parenting.

Name	Type	Size	Description
Oracle_User	varchar2	32	The name of the oracle user. This is always available.
Proc_Name	variable	65	The name of the oracle procedure that logged this run.
Last_Run_Time	Date		The time and date when this process last ran.
Next_To_Last_Run_Time	Date		The previous value. Useful in calculating the spacing between runs.
Next_Run_Time	Date		The date and time when we next expect to see this run. This is the key trigger for notification.
Run_Comment	varchar2	255	An optional comment set by the caller that will be displayed in status messages. Unlike the Error_Flag, this does not trigger notifications.

Table 3: Process_Monitor table – reporting.

machine. For example, we have a routine that we run daily to compare the Simon_Banner_Students table with the student base table (SGBSTDN) on our administrative machine. This is typical of many similar routines, and it has two optional parameters, a Target_PIDM which allows us to update a record for a specific person, and a StopCount which stops the update after a set number of records (this is handy for debugging). When neither parameter is set, we want to record the fact that the routine ran to completion.

In Figure four, we have a code segment of the routine that checks the student base table on Banner (our student record system) and updates the Simon student table. The two cursors are written so that if they are opened with a value for the PIDM, they will return just the single record for that person; otherwise, they will return a full set of records. There is also an option to stop after a set number of rows. Once the loop is complete, and if we did *not* exit due to the stop count, and we were *not* doing the check on behalf of a specific individual, we will record this run using the Process_Monitor_Record.Mark_Proc procedure. This procedure will obtain the user, hostname, and other information from the database environment. The only thing we need to give it is the task name (Target_Name) and the name of the current package. In this way, whenever we do the general update, it will record the fact that it ran; it doesn't matter how we did it.

Generate_File Definition

A number of the tasks that we want to watch are run via our Generate_File system (described in the LISA 2000 Proceedings). These tasks are generally reading or writing files, using stored procedures in the

```
procedure Sgbstdn_Full(Target_Pidm in number,
                      stop_count in number)
is
    Banner      Sgbstdn_Scan_Curs%RowType;
    Simon       Simon_Scan_Curs%RowType;
    Act_Cnt     number := 0;
is
    Open Sgbstdn_Scan_Curs(Target_Pidm);   -- Full scan if NULL
    Open Simon_Scan_Curs(Target_Pidm);   -- Ditto
    loop
        . . . (Details of processing omitted)
        exit when Act_Cnt > Stop_Count;
    end loop;
    close Sgbstdn_Scan_Curs;
    close By_Pidm_Curs;
    if Act_Cnt > Stop_Count
    then
        dbms_output.put_line('Stopped due to stop_count');
    elsif Target_Pidm is null
    then
        Record.Mark_Proc(Target_Name => 'Student-Sgbstdn',
                          Procedure_Name => 'BStudent_Maint');
    end if;
end Sgbstdn_Full;
```

Figure 4: Recording run from a PL/SQL procedure.

database. When we develop a new file target, we store the PL/SQL source code in a file, and read that into the database using SQL*PLUS.² At the end of this file, we include a block of PL/SQL to register the new targets with the system. This is done with a procedure called Add_Target_Simple or Add_Target_Complex.³

In Figure five, we have a fragment of the file used to generate some web pages documenting our network routers and subnets. In the package, we define some entry points that will be called by the Generate_File system. At the end of the sample, we register three things: a simple target (web_routers) that will call the Get_Router_Html routine to generate a list our our routers into the primary_routers.html file, a complex target that will generate a set of files based on Get_Network_List routine, and, finally, a special target, Add_Process_Record, that will record the fact that the first two entries have been executed and have completed. This last routine makes it trivial to record the completion of any Generate_File run by simply adding the Generate_File.Add_Process_Record to the end of the registration statements.⁴ This has the added advantage of not having to modify the source code that doing the direct PL/SQL call would require.

The Add_Process_Record routine actually calls the Add_Target_Simple routine to register a special

²SQL*PLUS is a command line interface to Oracle. One of the options is to read from a file and pass the information to Oracle for processing.

³Since the original paper, we have added several other kinds of targets in addition to these.

⁴The Generate_File registration routines will default to the target name specified in earlier calls if not provided on later calls.

target that just records the fact it was called, and exits after writing a few lines to stdout. Since it is called from within the Generate_File environment, it can get all of the information it needs for recording from that, and we don't need to pass in any parameters. It also prepends GENERATE_FILE- to the target name to come up with the name to record.

Special Generate_File Target

We still have tasks that we are interested in watching that are not written in PL/SQL or using Generate_File. These might be older file generation programs, or just shell scripts run out of cron. The Generate_File program has the ability to pass a parameter to the processing routine. We combined this, with a variant of the previous routine to have a new Generate_File target that will record anything, with a prefix of MANUAL-.

In Figure six, we have a simple shell script that is run from cron to generate the /etc/printcap file for our

```
define name=GENERATE_NETWORK_LIST
prompt Create Package &NAME
Create or Replace Package &NAME as
--
-- Generate web pages documenting our network.
-- Define the standard interface
procedure Get_Network_List(Fname out varchar2, Dbmsout out varchar2);
Procedure Get_Router_Html(result out varchar2, p1 in varchar2, p2 in varchar2);
Procedure Get_Subnet_Html(result out varchar2, p1 in varchar2, p2 in varchar2);
... Package definition omitted
end &NAME;
/
begin
Generate_File.Add_Target_Simple(
    target => 'web_routers',
    filename => 'primary_routers.html',
    get_data_rtn => '&NAME..Get_Router_Html');
Generate_File.Add_Target_Complex(
    get_attr_rtn =>'&NAME..Get_Network_List',
    get_data_rtn =>'&NAME..Get_Subnet_Html');
Generate_File.Add_Process_Record;
end;
/
```

Figure 5: Recording Run from a Generate_File target.

```
#!/bin/sh
#
# Script to Generate /etc/printcap
#
FILE_GEN=/campus/rpi/simon/directory/2.0/@sys/bin/Generate_File
PCAP_GEN=/campus/rpi/simon/printmaster/1.0/@sys/bin/etcprintcap
#
if [ -x $PCAP_GEN ]; then
    $PCAP_GEN
    if [ $? -ne 0 ]
    then
        echo "Error in printcap generation!!!!"
        exit 1
    fi
    $FILE_GEN -target Record_Process -par2 Printcap
fi
```

Figure 6: Recording run from a generic Generate_File target.

system. Assuming that the program exists, and runs successfully, we then call Generate_File with the target Record_Process to record the completion of the task MANUAL-Printcap.

Notifications

Although it is all well and good for the database to know when a process is overdue, we really need some way of letting the appropriate people know about this. It is important, however, that the mechanism used is appropriate for the type of failure and the urgency of the process. For example, when the process feeding password changes into our Active Directory server fails, we want to get the service restored within minutes. But if the billing run for our backup service is a day late, it isn't a major problem; we normally run this two or three times a year.

Most of the tasks we are monitoring run once or twice a day. As a result, we are currently only

checking for “late” tasks a few times a day, generating a report, and mailing it to interested parties. At present, we don’t have anything in place to escalate a problem that has not been repaired in a timely fashion. So far, the notifications are unique and infrequent enough that they are not ignored.

In Figure seven, we have a notification email from the system. This is actually sent as an HTML page, instead of a plain text message. While this might be annoying to some, I already had the code to generate the late list as a web page in the administrative tool, and I just had to wrap it in a call for `Generate_File`. This has the added advantage that the buttons visible on the email message are fully functional, in that I can press one and see the detailed information for that task. In this case, I have three tasks that are late, printing and disk billing (that was desired actually, as we were deferring some revenue to the new fiscal year) and the `Generate_File` run that produces our Building Directory web pages. In this case, the normal run had failed due to the administrative database being down for a backup.

Conclusions

The Task Monitor tool has proven to be very useful in detecting things that should be happening and failed for some reason. This is especially useful for the infrequent jobs that are easy to forget. Because it is so easy to add monitoring to existing tasks, the number of things that we watch has grown quickly.

Existing Limitations

Not all aspects of the original design have been implemented. At present, we are only checking for “late” processes twice a day. Before we can make this a more frequent occurrence, we need to implement the notification limits. While it may be useful to check for late processes every two minutes, I don’t want to get an email every two minutes when a monthly task is a day late.

Another part we have not implemented is dealing with non regular, but recurring schedules. A number of our business applications do not run on the weekends. The intention to handle these would be to support cron style schedules, and figure the next run time based on the

Name	Type	Size	Description
Contact_List	varchar2	128	A list of email addresses to contact when problems are detected.
Error_Flag	varchar2	128	An optional error message set by the process. Results in immediate notification.
Notify_Delta	Number		The minimum time in seconds between notifications when an error has been detected.
Last_Notify_Time	Date		The time and date when notification was last attempted.
Next_Notify_Time	Date		The time and date when notification will next be attempted if a successful run has not occurred.

Table 4: Process_Monitor table – notifications.

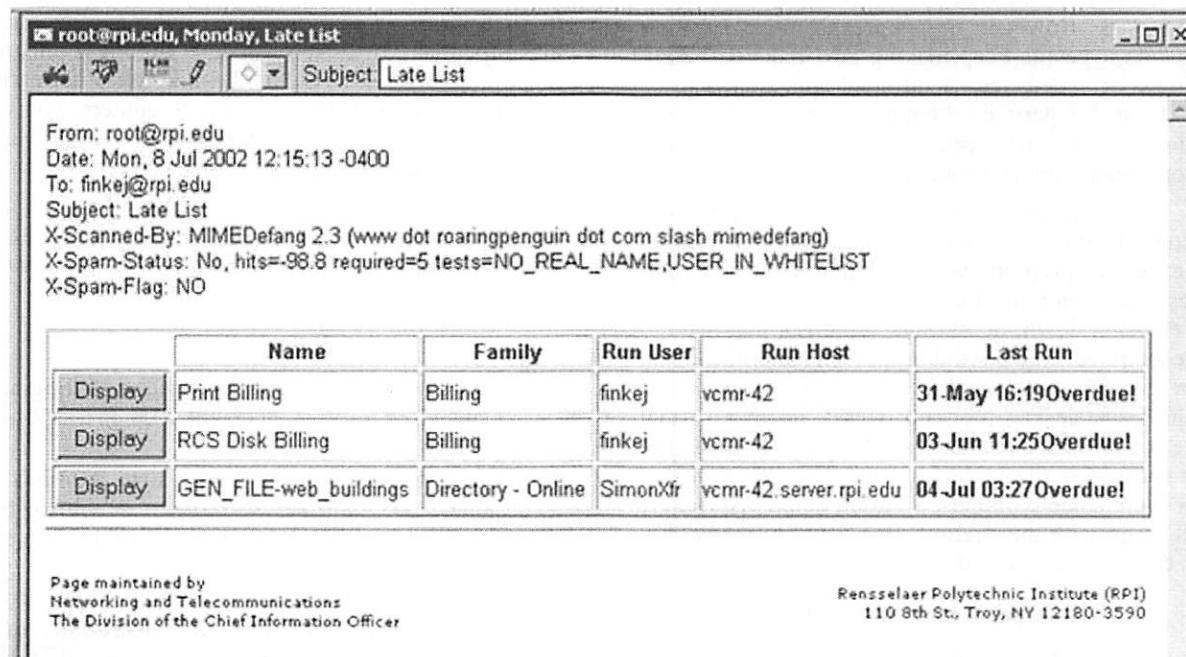


Figure 7: Sample “late” message.

cron format schedule plus the run delta. This would also make it easier to handle manual operations that we expect to be done each business day. This still does not handle holidays; this needs some more thought.

Future Directions

All of the tasks we are currently monitoring with this system are either directly accessing the database, or have the `Generate_File` program available. However, in order to help track activity on other (Unix) systems, a syslog or SNMP interface to allow other things to occasionally report in might be very useful.

As the number of system specific tasks, such as the last run of CFEngine [1] on a machine, grows, some automatic classification and run delta assignment would remove the bottleneck of having to assign an owner, family, and schedule information for each new service. The ability to designate a particular task entry as a “prototype” for new tasks of the same name and different system identifier would make this very easy.

Other notification methods need to be explored. The ability to generate syslog or SNMP messages and direct them to other monitoring tools could be very useful. This could in turn generate pages, or be directly incorporated into this system.

Another extension to this project is as basic reminder system. This would just require a tool to manually enter a new process, and directly set the `Next_Run_Time` value. This might be used to remind people to reset annual allocations or renew licenses and service contracts.

We also have a number of tasks that are started on response to some user request, such as a quota change request. One approach would be to have the request also set the “next run” time for the quota change task. However, this approach might run into problems if people keep making new requests before the timeout is detected. A different approach is to be able to make periodic “empty” requests that will require that the task finish all queued work. Both options need some consideration.

Several of our file generation scripts are run out of cron. These are basically shell scripts that run the `Generate_File` program with different targets. Sometimes one or more of these will fail, possibly due to a server being down, or PL/SQL packages that need to be recompiled. One possible approach would be to add a “rerun” flag to the shell script that would be passed to the `Generate_File` program. If set, another special target could be added that would have `Generate_File` skip the run if the target was not late. With this, if there were some problems, a person could just run the shell script with the “rerun” flag, and only those targets that were late would get regenerated.

References and Availability

All source code for the Simon system is available on the web. Please refer to the following URL for

details: <http://www.rpi.edu/campus/rpi/simon/README.simon>.

In addition, all of the Oracle table definitions as well as PL/SQL package source are available at <http://www.rpi.edu/campus/rpi/simon/misc/Tables/simon.Index.html>.

Although this is implemented in Oracle as part of the Simon system, there is very little that requires Simon or even Oracle. Just about any relational database would be able to handle the moderate processing and database needs for this system. Given our starting point, most of our examples are deeply tied to Simon, but with alternate interfaces such as syslog and snmp, there is no reason why this could not be deployed without Simon or Oracle.

Acknowledgments

I would like to thank Marcus Ranum for his shepherding of this paper, as well as Deb Wentorf for her proofreading and editing. I also want to thank Rob Kolstad for his excellent (as usual) job of typesetting this paper.

Author Biography

Jon Finke graduated from Rensselaer in 1983, where he had provided microcomputer support and communications programming, with a BS-ECSE. He continued as a full time staff member in the computer center. From PC communications, he moved into mainframe communications and networking, and then on to Unix support, including a stint in the Nysernet Network Information Center. A charter member of the Workstation Support Group he took over printing development and support and later inherited the Simon project, which has been his primary focus for the past 11 years. He is currently a Senior Systems Programmer in the Networking and Telecommunications department at Rensselaer, where he continues integrating Simon with the rest of the Institute information systems. When not playing with computers, you can often find him building or renovating houses for Habitat for Humanity, as well as his own home.

Reach him via U. S. Mail at RPI; VCC 319; 110 8th St; Troy, NY 12180-3590. Reach him electronically at finkej@rpi.edu. Find out more via <http://www.rpi.edu/~finkej>.

References

- [1] Burgess, M. “A Site Configuration Engine,” *Computing Systems*, 8(1):309, MIT Press, Winter 1995.
- [2] Burgess, M., “Theoretical System Administration,” *The Fourteenth Systems Administration Conference (LISA 2000)*, p. 1, USENIX, December 2000.
- [3] Finke, Jon, ‘Automating Printing Configuration,’ *USENIX Systems Administration (LISA VIII) Conference Proceedings*, pp. 175-184, USENIX, September 1994.

- [4] Finke, Jon, "Monitoring Usage of Workstations With a Relational Database," *USENIX Systems Administration (LISA VIII) Conference Proceedings*, pp. 149-158, USENIX, September 1994.
- [5] Finke, Jon, "Institute White Pages as a System Administration Problem," *The Tenth Systems Administration Conference (LISA 96) Proceedings*, pp. 233-240, USENIX, October 1996.
- [6] Finke, Jon, "Automation of Site Configuration Management," *The Eleventh Systems Administration Conference (LISA 97) Proceedings*, pp. 155-168, USENIX, October 1997.
- [7] Finke, Jon, "An Improved Approach to Generating Configuration Files from a Database," *The Fourteenth Systems Administration Conference (LISA 2000)*, pp. 29-38, USENIX, December 2000.
- [8] Finke, Jon, "Embracing and Extending Windows 2000," *The Sixteenth Systems Administration Conference (LISA 2002)*, USENIX, November 2002.
- [9] Gilfax, M. and Alva Couch, "Peep (The Network Auralizer): Monitoring Your Network with Sound," *The Fourteenth Systems Administration Conference (LISA 2000)*, p. 109, USENIX, December 2000.
- [10] Hansen, Stephen E. and E. Todd Atkins, "Automated System Monitoring and Notification with Swatch," *USENIX Systems Administration (LISA VII) Conference Proceedings*, pp. 145-156, USENIX, November 1993.
- [11] Hoogenboom, P. and J. Lepreau. "Computer System Performance Problem Detection Using Time Series Models," *USENIX Systems Administration (LISA VII) Conference Proceedings*, p. 15. USENIX, November 1993.

An Analysis of RPM Validation Drift

John Hart and Jeffrey D'Amelia – Tufts University

ABSTRACT

Experiments that analyze dependencies in RedHat Linux and RpmFind.net show disturbing conflicts and overlaps between software packages that result in installing multiple differing versions of dynamic libraries. The final state of a system containing conflicting packages depends upon the order in which packages are installed, as well as user input during the installation process. This leads to system states that may or may not have been tested, lowering confidence that the resulting software configuration will function properly. We describe the details of the problem, potential effects, and potential solutions involving improving the practice of building RPM packages.

Introduction

RedHat Package Manager (RPM) files and their equivalents have revolutionized the ease with which one can add software to a Linux system. But do RPMs embody ease, or perhaps danger? The following excerpt from Ladislav Bodnar's article "Is RPM Doomed?" [2] gives a very accurate account of a situation that most system administrators have experienced:

"You have just found this great software on the Internet and off you go to download and install it. It's all free and GPL and, as luck would have it, the author provides a binary package in RPM format. It doesn't take long to download it, then you run the customary rpm -Uvh package-name.rpm command. OOPS! The installation fails, reporting a missing dependent package without which it will not install or function correctly. Off you go again to search the Internet for the missing library.

Unfortunately, installing that missing library fails because of three other missing libraries and two other libraries that come in incorrect versions. Depending on how badly you want the original package, you have two choices – either go and search for all missing dependent libraries as well as all libraries dependent on the dependent libraries, or you just give up. How many times have you given up?

If you are persistent and lucky, you might eventually install the RPM package. If you are persistent and not lucky, then you have probably acquired a few bumps from banging your head against the nearest wall in sheer desperation. RPM dependency hell can be a hugely frustrating experience – anything from circular dependencies (the catch 22 situation) to incorrect library version when there wouldn't be much left untouched had you really persisted in getting that badly wanted RPM installed."

The root of several problems with RPM (and many other kinds of package management) is that "order matters" [11]. It is common practice among

many administrators to install and uninstall RPMs and other kinds of software packages with little concern for change control and without keeping a journal of the order of modifications. But case studies and theoretical analyses [10, 11] suggest that the only way to produce a predictable and reliable system is to decide upon some particular order for package installations and other configuration actions, and always perform the actions in that order. Another independent analysis [5] suggests that order matters whenever system configuration actions do not take a rather restrictive form in which all configuration actions are "homogeneous" with one another; this means that if two actions change the same file, they change it to have the exact same content. While this would seem a reasonable requirement, in our experience, RPM installations do not satisfy this restriction.

How dangerous is it in practice to ignore this discipline of ordering? It seems from practical experience that the danger is far greater than most of us realize. Many of us have managed to put systems into a state where "only starting over is feasible." Why is this so?

This study looks at the risks associated with installing and uninstalling RPMs. We look at the nature of dependencies between packages to understand how one package has the potential to break another. We explain how use of poorly structured RPMs causes a "validation drift" in which the final system gradually "drifts" over time to a configuration that has not been tested. Using global analysis of existing RPM repositories, we identify subtle inconsistencies in well-known RPM packages that can lead us to doubt the results of installing them.

First we must comment that this study is limited in several ways. We only study the i386 distributions of RedHat 6.2, RedHat 7.2, and the Contrib directory, as listed on RedHat.com and RpmFind.net. These directories contain highly volatile data that will be quickly outdated. Much of the work was done in April of 2002 and repeated in July of 2002, with differing results due to changes (mostly improvements) in

repository files! We are happy to report that one major example of repository rot that was discovered in April could not be reproduced from July data. We hope that *none* of the inconsistencies we report will ever be reported again, because implementors and repository managers will be motivated to address the problems we have found. Though there may be different inconsistencies, do not expect to necessarily find any of these particular problems in future RPM repositories.

Why RPM?

There are many package managers available to Linux developers and choosing one to focus on was difficult. Package managers such as Debian's DEB [13] or Slackware's TGZ [15] would have been fine choices for our analysis. However, the nature of RPM makes it a good basis to analyze the nature of installation failures and validation drift while also providing a framework for thinking about solutions to these problems. RPM is deployed in large and small network installations and many system administrators depend on it for installing and maintaining complex sets of software. It has a rich feature set that allows the installation and removal of individual packages or sets of packages and it maintains an internal database that records and verifies each change to the system.

Like most other package managers, RPM as a system allows a user to install packages in a computer, while checking an internal database to verify that all known prerequisites are met before allowing the package to install. After configuration, RPM allows the package to run arbitrary binary and script files prior to and for the completion of installation and uninstallation [1]. Because of this, we concentrate upon the Red Hat package management system [1] for Red Hat 6.2 and 7.2, including the contributed packages available from <http://www.rpmfind.net>.

Many readers have commented that we already know that the contrib directory is broken, so why analyze it? We respond that it helps to know *how* things break, so that we can avoid them in the future. We knew when we started that there would be serious problems, but had no idea of the nature of the problems we would find. And, surprisingly, the problems we found are not the problems that we expected!

RPM Dependencies

In every RPM package there exist several different kinds of dependencies. Declared dependencies external to the file are contained in the header information in each RPM package. Each package declares which services it "provides" and "requires." A service is nothing more than a string. RPM satisfies dependencies by forcing one to sequence software installations so that services are "provided" by installed packages before they are "required" by others.

But also, each executable file in an RPM archive has requirements, some of which can be determined

through use of techniques like those of sowhat [4]. These internal dependencies are intrinsic to the file and may or may not be related to dependencies declared in the RPM header. Sowhat's analysis utilizes the output of ldd and is not exhaustive; one can subvert it, e.g., by using dlopen to open a dynamic library by name, bypassing ldd and ld.so.conf.

Some kinds of errors are relatively insignificant. If an RPM package is over-declared (the set of dependencies in the header exceeds the set that the program needs), the consequence is that extra, possibly useless, programs are installed. If a package is under-declared, the packages actually used and required are greater than what is declared. This means that a package installation will fail even if the declared dependency requirements are fulfilled.

Are RPM Dependencies Sloppy?

The root of all evil in RPM seemed to be – at the outset – the way RPM packages are expected to declare what they need in order to operate. In RPM, there is a simple mechanism for notating dependencies between packages. In the package header, each package is declared to "provide" zero or more services. These are just strings with no real semantic meaning. A package that needs a service then "requires" it. This mechanism is mainly used to declare dependencies between packages using a dynamic library and the package(s) that might provide a copy of the library, so the "services" are typically library base-names.

Those of us who have experienced "dependency hell" (as documented in the excerpt in the introduction) have suspected major problems with the RedHat dependency system. We attempted to validate our suspicion by running a simple test to check the difference between actual and declared dependencies. The results were both more encouraging than expected and, in a way, depressing. The true dependency errors seem to be few in number and cannot account for the trouble many people report in using RPMs.

Checking Dynamic Library Dependencies

We could not in general determine all dependencies for a package, but we can determine all the dynamic libraries needed by a package. We did this by unpacking each package (using cpio) and skipping execution of the installation scripts. Then we ran ldd on each executable or dynamic library in the package. Finally, we compared the actual dependencies exposed by ldd with the declared dependencies in the header.

Comparing these was a complex process due to the free-form nature of dependencies. It was a multi-step process that takes into account all the ways a dependency can be declared in a collection of RPMs. The cases for each target RPM (the RPM from the collection currently under investigation) include:

- a) The required library is part of the target RPM that requires it. In this case, no dependency listing is required.

- b) The required library is explicitly required by the target RPM and provided by another. This is normal.
- c) The required library is part of an RPM that provides *another* service that happens to be required by the target RPM. This is an implicit dependency based upon a service tag that is actually unrelated to the real library dependency (Figure 1). This is usually bad style for dependency declarations, at least for dynamic libraries. The only exception is that to save space, some implementors use blanket tags for service subsystems, e.g., “require qt”. This is to avoid listing all the core libraries of the service explicitly.
- d) The required library is part of the RedHat core distribution, for which dependencies are not explicitly listed, as their presence in any RedHat system is assured.
- e) The required library is in another RPM with which the target RPM shares no explicit or implicit dependency. This is a dependency error.

To analyze the distribution of these kinds of dependencies within RPM repositories, we wrote two programs. Rift lists all of the dependencies in a set of packages that can be exposed through ldd. Tree reads all RPM files in an RPM distribution and outputs all dependency and checksum information from the distribution. The output of tree, together with the output of rift, is fed to a new program deps that categorizes dynamic library dependencies into each of the five classes above.

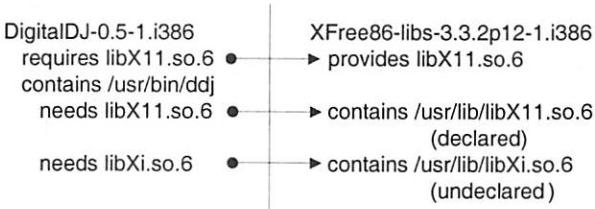


Figure 1: Implicit dependency of ddj upon libXi.so.6 via libX11.so.6.

Our results for the RedHat i386 distribution are shown in Table 1. The good news is that the distribution itself, as it comes from RedHat, is rather well-constructed with few errors. The errors seem to be statistical outliers. Errors we found were isolated to two packages. The anaconda runtime package anaconda-runtime-7.2-7.i386.rpm fails to require lib-linux.so.2, libc.so.6, libresolv.so.2, and libz.so.1. These are part of the core distribution so that this has no observable behavioral effect; it is just a bit sloppy. PyQt-2.4-1.i386.rpm fails to require libstdc++-libc6.1-1.so.2; this is a bit more serious and requires user intervention.

Our results for the RedHat contrib directory (i386) are shown in Table 2. Here things become more “interesting.” Most packages are astoundingly well-behaved

about declaring their needs. Outright errors are almost a statistical outlier. Errors we observed are listed in Table 3. These are annoying but minor at best.

# deps	Kind of Dependencies
741	normal: “requires” and “provides” correct.
26	internal: package contains library upon which it depends.
5	errors: dependency declaration omitted.

Table 1: Dependency types in RedHat 7.2.

# deps	Kind of Dependencies
1201	normal: “requires” and “provides” correct.
21	internal: package contains library upon which it depends.
9	implicit: unrelated dependency includes file.
8	errors: dependency declaration omitted.

Table 2: Dependency types in RedHat Contrib.

Package	Fails to require
Eterm-0.8.8-1.i386.rpm	libungif.so.4
Frodo-4.1a-1.i386.rpm	Logo
ImageMagick-4.2.7-1.i386.rpm	libbz2.so.0
ImageMagick-perl-4.2.7-1.i386.rpm	libbz2.so.0
Qtabman-0.1-1.i386.rpm	libclntsh.so.1.0
XITE-3.3-3.i386.rpm	libjpeg.so.62
XITE-3.3-3.i386.rpm	libz.so.1
aktion-0.2.1-1.i386.rpm	libstdc++-libc6.1-1.so.2

Table 3: Dependency errors in RedHat i386 Contrib.

One disturbing tendency was some use of implicit loading of libraries. There were nine instances in which a dynamic library was required implicitly as a side-effect of another explicit requirement. The X11 library libXi.so.6 was implicitly required six times as a result of explicitly requiring libX11.so.6. Likewise, libdl.so.9 was implicitly required three times as a result of explicitly requiring libc.so.6. These libraries were also among those that were formerly bundled with the libraries whose dependencies load them. The implicit loads of these libraries are probably due to packages being designed before that library design change took place.

The prognosis of this work is surprisingly good. While it would seem that the contributed RPM repository would be chaos, our simple checks showed contributed RPMs to be relatively organized and well-structured. Our obvious question, then, is “what is really wrong?” It is *not* the dependencies, because

errors in these are statistically insignificant. We must look deeper for potential problems within RPMs. The key to this looking deeper is the concept of *validation* of the resulting system.

Validation

The central theme of this paper is not dependency analysis itself, but rather the relationship between dependency analysis and validation of software installation. In software engineering [9], there are two forms of sanity checking:

- “verification”: “are we making the product right?” Does it conform to our own ideas of how it should work?
- “validation”: are we making “the right product?” Does it conform to customer needs?

In system administration, we often concentrate upon validation to the exclusion of any concept of verification. We have less design freedom than software authors, and user requirements are usually more completely spelled out than for a software engineer, so that there is much less difference between “verification” and “validation” than there would be in a software development environment. The key to validation is rigorous testing [12] in a realistic setting. One must actually try the system and see if things work as expected.

Last year, Couch and Sun described global analysis [4] without emphasizing validation, its most important component. Any analysis of what went wrong with a system must be tempered by knowledge that at some time in the past, “things were right.” One’s reasoning must always flow from knowledge that the system did work properly before. Otherwise, the question of “what broke” – central to use of sowhat – has no meaning. Unfortunately, it is often the case that the user thinks “things were right” in their system configuration when in fact they have been working with a system that was not completely validated. This perception by the user can contribute to the problem at hand. Dependency problems are always problems of validation: “can we be sure that in making a change, we do not break anything?”. Given a rather strongly validated core distribution of RPMs, how can we avoid breaking anything in it that worked before?

Validation Rot

Brooks [3] points out that in software engineering, “software rot” occurs when too many small changes are made to a complex system, so that no one really understands the function of the software. Couch points out that a similar kind of “filesystem rot” occurs in software repositories managed over long time periods [6], so that meanings of specific files become unclear and inaccessible.

Our analyses show that RedHat machines managed via RPM suffer from a new kind of rot: “validation rot.” This is a gradual divergence from a fully

tested configuration that invalidates and undermines prior testing. Here is how it works:

- We start with a “baseline configuration,” e.g., a RedHat distribution. This configuration is present on a multitude of hosts around the internet, so we can wait until this baseline has been comprehensively validated by an extensive community of users.
- Gradually, over time, we add functionality in the form of “contributed RPMs.” Each one of these adds some files and may replace others. The result is that the system diverges not only from the baseline, but also into a fairly unique state that may not be replicated anywhere else on the Internet.
- At any point in this process, one has a unique system that has never been validated by anyone.

The user community (and vendors) validate packages in relation to the baseline, not in relation to other packages. It is nearly impossible to test, yet alone reach, all possible package states due to combinatorial explosions. This means that a user is “at risk” when their system reaches a configuration state that has not been achieved by any previous population of users or the software developer. It is possible, then, that there are latent bugs that will show up only on that particular machine.

Validation Expense

Validating software is expensive. For commercial software, it often requires the expertise of a Software Quality Assurance(SQA) [9] team. For open-source freeware, the user community itself often serves that purpose over longer time periods, submitting bug reports and fixes. Either way, software can only be validated as working properly by extensive testing in multiple environments and with various kinds of inputs. There is no such thing as “completely tested software” [10] and one must always decide what form of testing is “good enough” or “complete enough” [12].

Transitive Validation

The expense of validation has led the Linux Standard Base [14] to employ a “transitive validation” strategy. Previously, a vendor wanting to market a software package for linux had to validate its function on every distribution of linux. The Linux Standard Base was created in order to give vendors the assurance that a package that works somewhere, works everywhere. If we control the couplings between a software package and its operating environment, and can validate the environment as possessing appropriate couplings, then validating it in one compliant environment validates it in all such environments.

There are two parts to the Linux Standard Base:

- 1) a code validator that indicates whether a specific binary file is compliant with the base. This

checks whether the system calls used by the binary file are loaded from correct versions of dynamic libraries.

- 2) an environment validator that checks whether the environment on a specific linux machine complies with the minimal requirements needed for system calls to work properly. This checks not only the existence and versions of key libraries, but also checks that particular system control files are found in standardized locations, e.g., `/etc/hosts`.

The key assertion chain of LSB is that:

- a) If a particular vendor software package passes code validation, i.e., only utilizes approved system and library calls, and
- b) The vendor package has been tested on one LSB-compliant system, and
- c) A particular linux system passes environmental validation, i.e., has all its libraries and files in appropriate places, then
- d) The vendor software should function fine on *any* environmentally compliant system.

This is a “transitive validation” claim: software that works in one compliant environment works in every such environment. This can potentially save tons of money in validating software for different distributions of linux.

Is Validation Trust Transitive?

We are inspired by the LSB strategy and would like to apply a similar process to the problem of validating RPM-managed systems. The key question is “What can we trust?”. Trying to answer this question cuts to the heart of the RPM problem. We contend that “we usually put too much trust in existing infrastructure.”

For example, let’s consider the following apparent assertions about “transitive trust”:

- 1) If a program works properly on a RedHat 7.2 system, it will work properly on all RedHat 7.2 systems.
- 2) If a script works properly for a particular version of Perl, then it will work properly in the same version of Perl regardless of the environment in which it executes.
- 3) If a program or dynamic library compiled with one compiler works properly, then it will work properly if compiled with another compiler.
- 4) If a program works before new software is installed, it will work after the software is installed.

In each case, we decide to trust something in a new situation based upon validation in an old situation. All of the above assertions seem reasonable, and all are quite obviously *false* to the point of being ludicrous. Each of these points can be expressed in realistic terms as follows:

- 1) Just what is a RedHat 7.2 system? This is the baseline, but what has been done to the system

since then? If a program depends, e.g., upon `/etc/foo`, then it will only work if one has installed `/etc/foo`. This has nothing to do with the baseline.

- 2) Any Perl programmer knows that Perl does some rather strange things to cope with system differences. For example, its implementation of lockf can take at least four forms depending upon support for locking in the operating system. These forms are semantically different.
- 3) Modern compilers have bugs, especially when optimization is turned on. Validation under one compiler is no guarantee of function when the same program is compiled with another.
- 4) Even in the simplest of cases, it is easy to break a program by installing another. The problem is “hidden dependencies” between programs and other programs and libraries.

These simple examples are obviously bogus, but administrators who install RPMs on an ad-hoc basis are using them as assumptions. We come to the inexorable conclusion that a system is validated as functional if:

- 1) it is constructed starting from a validated baseline,
- 2) all software packages installed in addition to the baseline are:
 - a) validated against the baseline configuration by being installed against it and thoroughly tested.
 - b) homogeneous [5], in the sense that overlaps between packages other than the baseline install the exact same content.
 - c) uncoupled from the contents of other packages (excluding homogeneous overlaps), so that software within each package only refers to baseline content and the content of the specific package.

Analysis of RPM Failures

There are four main things that can interfere with the proper operation of a single package:

- 1) Hidden dependencies not known to the package designer.
- 2) Version skew between files and the programs that utilize them.
- 3) Relationships between files that are obscured by scripting.
- 4) Asynchronous operations other than package management that affect package files or required files.

Hidden dependencies are those unknown to the package designer or simply undeclared. Every package implicitly depends, e.g., upon the whole base distribution. If something in the base distribution changes, the package may break, but such dependencies are never made explicit. According to the results above, though, these may be statistically insignificant.

Version skew is a very common problem with which many people are familiar in both the Unix and Windows worlds. This happens when a library or program associated with more than one program is upgraded and the newer version is not functionally compatible with the older version.

Installation scripts, which are commonly included with programs today, are usually designed to move files and create directories that are custom to the package. But with larger, more complex packages, installation scripts are non-trivial and can perform tasks that have system-wide effects. Since the changes that an installation script can make are limited only by the rights of the user running it, (the user is typically root) any program has the ability to touch any other program or file. A common example is when an Apache RPM is installed. It makes modifications to the `inetd.conf` file that are not obvious if one is not aware that modifications are being made.

Asynchronous operations, which include installing non-RPMs, manually changing files that rpm controls, hacking, etc., can also have a compelling effect on the validity of installations. Since most complex systems span multiple volumes, when a package is located on one volume and a dependent package is located on a different volume, both volumes must be mounted or the dependent package may not work. This type of dependency is difficult to properly diagnose.

Global Analysis

The problem with the above descriptions of failure modes is that they are all module-centric. They can explain what's happening when one module is installed, but do not depict potentially subtle multi-module interactions. We applied the global analysis techniques of sowhat [4] to this problem and found potential and subtle failures in adding RPMs to the standard distributions. While the configuration language for RPM files allows expression of "backward" dependencies between referrer and required resource, "forward" dependencies between a new resource and an old program that uses it can lead to systems whose function has not been properly tested.

Our Experiments

We undertook several experiments to understand the scope of the problem of validation drift. Our first task was to identify the scope of inhomogeneity within RPM packages. We obtained several package distributions (for the i386 architecture) using the sites redhat.com and rpmfind.net. We then wrote several custom scripts to analyze their contents and pinpoint potential validation problems.

Inhomogeneities

Because of the computational difficulty of reading large distributions, we broke our analysis into several short scripts, each of which provides input to the

next. Our first Perl script tree reads all RPM files in an RPM distribution and outputs all dependency and checksum information from the distribution as a text file. This data is then read by a second script, `munch`, which computes a list of files that are inhomogeneously provided by more than one package, as indicated by differing MD5 signatures for the exact same file path. We then went over these inhomogeneities using a filtering script `punch` to eliminate conflicts based upon hardware differences (i386 vs. i686) where needed. This was done based upon the naming conventions for RPM files.

Results of this process differed greatly depending upon where we tried to do it. RedHat 7.2 (i386) exposed no conflicts whatsoever. RedHat 6.2 (i386) exposed 14 conflicts, of which one was a difference between a software bundle and an individual package; three were due to packages that are mutually exclusive, e.g., mail delivery agents; six were due to multiple versions of the same software, and the remaining four were due to unforeseen and simple packaging mistakes. Each "conflict" is a system file that has multiple versions listed in the RPM repository, where there may be up to five versions available for a single conflict.

But it should be no surprise that as a result of performing this process on the contrib directory, we found 3499 inhomogeneities distributed as in Table 4. The majority of the problems were due to the presence of multiple versions of the same software, sometimes with recognizable naming patterns, sometimes not.

# Errors	Kind of Error
3095	differing versions of the same software.
238	file version conflicts in apache modules.
80	software packages are mutually exclusive by design.
52	software bundle disagrees with individual tool package.
25	inconsistent versions in overlapping software bundles.
9	other conflict

Table 4: Inhomogeneities found in contrib directory of rpmfind.net.

Apache contributed modules were a source of great chaos; conflicts encompassed everything from HTML and GIFs to dynamic libraries as described in Table 5. One big surprise is that a single apache add-on, `php`, was responsible for 125 of the 238 file version conflicts for apache modules. Most of this was due to replacing – for no reason apparent to us – much of the HTML documentation for apache itself inside `apache_php3-1.3b6-1.i386.rpm`. This is a classic case of "repository poisoning;" one RPM creates inconsistencies that affect several others.

Another much more potentially serious problem is that there were 36 dynamic libraries with multiple

versions, as listed in Table 6. These are the libraries loaded by Apache httpd itself. This was due to the contents of only five modules as described in Table 7. Each of these modules contained copies of between 32 and 36 dynamic libraries.

# Conflicts	Kind of Conflict
113	HTML documentation (.html)
36	dynamic libraries (.so)
32	manual pages (.1-.8)
30	header files (.h)
10	executables
4	configuration files
3	other

Table 5: Kinds of file conflicts in Apache.

The remainder of the inhomogeneities were due to several kinds of problems. Several packages were mutually exclusive by design, e.g., mail delivery agents or service daemons for the same service. 52 times, a software bundle disagreed with the package containing an individual tool added to the bundle. 25 files were inconsistent among two or more different software bundles. nine conflicts were simply unforeseen couplings between files and modules, e.g., expect-5.31.2-2.rh6.1.i386.rpm surprisingly instantiates /usr/bin/ftp along with socks-4.3.beta2-2.i386.rpm.

Binary Differences

At a more detailed level, while executables and libraries that are exact binary copies of others are functionally identical, executables and libraries that exhibit *binary* differences may or may not be interchangeable. Binary differences are evidence that there may be a functional difference, but this functional difference may or may not exist when the programs are executed.

Binary differences between files can also occur for completely gratuitous reasons. One can use two different compilers to compile the same .c file to get two object files that are different in binary but identical in text *and* function. As indicated by the above analysis, validation of code is not invariant of choice of compiler.

Our goal was to look at the grouping of RPMs available to us and determine which executables and libraries showing binary differences were possibly problematic and not caused by gratuitous metadata. The basic issue is whether two files that differ do so in a way that changes the behavior of programs. The files upon which we concentrated are all the Extensible Link Format [8] files in a linux system, including executables and dynamic libraries (.so).

We compared two different versions of the same file through a C program (provided by our advisor Alva Couch) that compares the binary contents of the text, data, and bss segments of an ELF [8] file. If two ELF files – executables, libraries, etc. – do not differ in text, data, and bss segments, then they are

functionally equivalent, even if they differ in other metadata such as date, compiler version, etc.

# Versions	Dynamic Library
5	/usr/lib/apache/mod_access.so
5	/usr/lib/apache/mod_actions.so
5	/usr/lib/apache/mod_alias.so
5	/usr/lib/apache/mod_asis.so
5	/usr/lib/apache/mod_auth.so
5	/usr/lib/apache/mod_auth_anon.so
5	/usr/lib/apache/mod_auth_db.so
5	/usr/lib/apache/mod_autoindex.so
5	/usr/lib/apache/mod_cern_meta.so
5	/usr/lib/apache/mod_cgi.so
5	/usr/lib/apache/mod_digest.so
5	/usr/lib/apache/mod_dir.so
5	/usr/lib/apache/mod_env.so
5	/usr/lib/apache/mod_example.so
5	/usr/lib/apache/mod_expires.so
5	/usr/lib/apache/mod_headers.so
5	/usr/lib/apache/mod_imap.so
5	/usr/lib/apache/mod_include.so
5	/usr/lib/apache/mod_info.so
5	/usr/lib/apache/mod_mime.so
5	/usr/lib/apache/mod_mime_magic.so
5	/usr/lib/apache/mod_mmap_static.so
5	/usr/lib/apache/mod_negotiation.so
5	/usr/lib/apache/mod_rewrite.so
5	/usr/lib/apache/mod_setenvif.so
5	/usr/lib/apache/mod_speling.so
5	/usr/lib/apache/mod_status.so
5	/usr/lib/apache/mod_userdir.so
5	/usr/lib/apache/mod_usertrack.so
4	/usr/lib/apache/libproxy.so
4	/usr/lib/apache/mod_log_agent.so
4	/usr/lib/apache/mod_log_config.so
4	/usr/lib/apache/mod_log_referer.so
4	/usr/lib/apache/mod_unique_id.so
3	/usr/lib/apache/mod_bandwidth.so
2	/usr/lib/apache/mod_vhost_alias.so

Table 6: Number of multiple versions of each Apache dynamic library.

# Lib's	RPM file
34	apache-fp-1.3.3-1.i386.rpm
36	apache-mod_ssl-fp2000-1.3.12.2.6.2-0.6.0.i386.rpm
36	apache-php3perl-1.3.12-3nosyb.i386.rpm
32	apache-ssl-jserv-1.3.2-2.i386.rpm
35	apache_modperl-1.3.6-1.19-1.i386.rpm

Table 7: Apache RPMs asserting conflicting dynamic libraries.

The result of this analysis was that of the 981 inhomogeneities in ELF format, only 13 of these turned out to be functionally equivalent, and 10 of these equivalences arose from two differing revisions

of the same package. The remaining three were unusual; it turns out that the two RPMs:

- apache-mod_ssl-fp2000-1.3.12.2.6.2-0.6.0.i386.rpm
- apache-php3perl-1.3.12-3nosyb.i386.rpm

have exactly equivalent copies of mod_unique_id.so, mod_log_referer.so, and mod_log_agent.so, for reasons we do not understand.

In summary, lack of equivalence is the rule. There were 968 inhomogeneities that were not able to be proven as functionally equivalent because they differ in text, data, or bss segment.

Update Skew

We know that RedHat validates the core distribution and each update, and have verified that their core RPMs are homogeneous in RedHat 7.2. But our analyses of contributed RPMs show that it is easy to *partially* update a system so that updated files are version-skewed with respect to one another.

In several cases, notably involving contributed versions of Apache, updates *overlap* in asserting new contents for particular libraries. RPMs assure that *backward* dependencies are satisfied (so that all libraries used by executables in the update are updated), but fail to update so that forward dependencies are satisfied. Any *other* pre-existing program that happens to use the same library is at risk of malfunctioning unless it is updated or validated against the new library as well.

In analyzing global dependencies in the RedHat distribution, updates, and contributed modules, we have observed two main kinds of failure of validation. Either an existing program is forced to use new libraries with unpredictable results, or the contents of a specific library depend upon installation order.

Case Study: Updating Apache

In Apache, the exact contents of a library depend upon the sequence of RPM installations. It is considered good practice to encapsulate apache updates, extensions, and modules into individual RPM packages, where each package contains an Apache module and all files that the module might potentially need [1,7]. This means, however, that many files in the main Apache package are duplicated among the update packages. If duplicated files are identical across all packages, there are few potential problems, but if they differ, we have reason to suspect that system states are attainable that have not been tested by anyone.

By a simple signature analysis, we found, e.g., that in the updates to RedHat 7.2, there are five different versions of /usr/lib/apache/mod_autoindex.so contained in five module RPMs:

- apache_modperl-1.3.6-1.19-1.i386.rpm
- apache-mod_ssl-fp2000-1.3.12.2.6.2-0.6.0.i386.rpm
- apache-ssl-jserv-1.3.2-2.i386.rpm
- apache-fp-1.3.3-1.i386.rpm
- apache-php3perl-1.3.12-3nosyb.i386.rpm

In principle, all copies of mod_autoindex.so should be identical in function, but comparison of md5 signatures

shows that all these files *differ* in binary content. This means that there are five different states for this file on an updated system, depending upon which updated module is installed *last*.

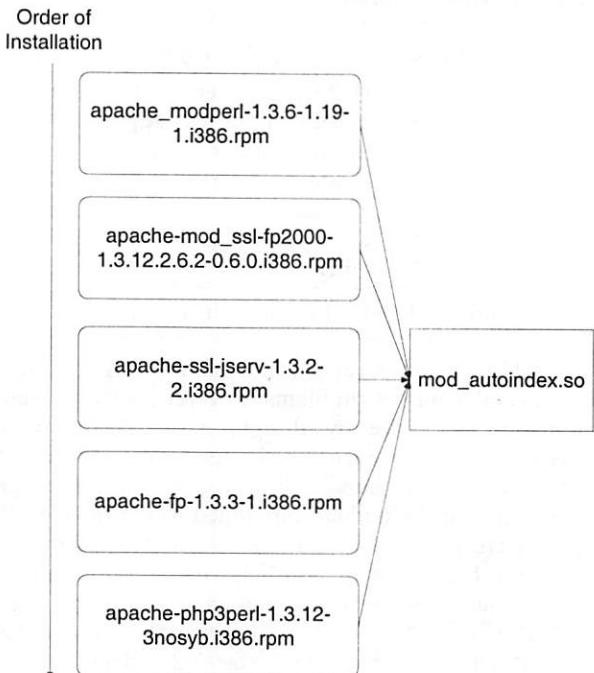


Figure 2: The unseen effects of updating apache: validation rot.

Figure 2 shows one possible configuration that could be used as an installation sequence for the five apache RPMs described above. The arrows from each RPM to mod_autoindex.so represent that RPMs installation of mod_autoindex.so. The problem here is that each RPM file has its own version of mod_autoindex.so. Thus, based on the order of installation and user input (about replacing files), the RPM that is installed last is the one that determines which version of mod_autoindex.so will be the one used by all of the packages once they are installed. This can cause the user to fall into an untested state and ultimately lead to a system failure.

Now in an ideal world, all that might differ in the various copies of mod_autoindex.so is “circumstantial” and does not affect behavior, e.g., the time that the source file was compiled. We actually *think* this is the case here, but have no easy way to validate contents against source code. This configuration situation represents a *risk* that one or more of these copies exhibits different behavior than the others. To be sure that a system works properly after updating, we need to know that the version that we have has been validated with the other modules that use this library.

Lessons Learned

Our preliminary analysis of the “contrib” branch of the RedHat distribution indicates that there are

roughly 324 potential library version skew conflicts, without even considering supporting executables and scripts. We believe that an even more detailed analysis will expose many more skew conflicts. These conflicts overshadow the dependency errors in RPM declarations, which are statistically insignificant by comparison. While these types of errors may be statistically insignificant, they can be annoying and can cause major problems in some instances.

Many administrators suffer from the illusion that one can install and manage packages in a relatively ad-hoc way at low cost. This illusion is shattered by considering the implications and cost of testing of ad-hoc configurations with the same rigor with which core distributions are tested. Testing is very costly, we consider it the vendors' responsibility, and yet we put our systems into states the vendor could not test and validate. If we fail to test the system ourselves, then the user of the system will inadvertently test it for us.

Using global analysis techniques, it is possible to predict whether one is moving to an untested configuration and to take corrective action so that one's system remains one that is widely used and tested. It is possible to minimize divergences between one's configurations and those used by the broader community, and to understand the cost of ad-hoc or divergent administrative methods.

Changing Practices

So what can we do differently to avoid these problems and assure that what we install will work properly? The key seems to be a different attitude and technique in creating and using RPMs. We can demand homogeneity in RPMs contributed by outsiders. We can analyze their dependencies and validate this homogeneity to some extent. We can minimize the effects of scripting by re-architecting RPMs and systems to have simpler script requirements, e.g., by creating directories rather than files, i.e., xinetd.conf rather than inetd.conf.

Avoiding gratuitous changes to validated baseline distributions will help ease the problems as well. We must look at changes more carefully before we make them. Each change not only represents a modification to our system but also pushes us farther from the baseline system. By constantly updating and changing our systems, we are moving farther and farther away from the baseline and a system that we know is validated. By viewing system changes as both updates and jumps from the baseline, we must make more informed decisions before we gratuitously install packages.

We can also avoid conflicts over dynamic libraries by making crucial libraries specific to the packages they serve. If a package needs a special dynamic library, name it differently than the normal one to avoid conflicts. This will help alleviate the problem of a library getting updated from one package when another package relies on the old version of it.

By coupling packages with the libraries that are crucial to their proper functioning, we can remove a problem that is difficult to recognize prior to system failures. This strategy trades memory for robustness; one must often make a library effectively unshared in order to isolate it from interactions.

Avoiding update skews in large packages by updating coupled executables and libraries simultaneously will continue to solve the problem of multiple packages relying on a single library. When several packages rely on an important library, an upgrade of any one of them can create unpredictable behavior. Even more problematic is the differing behavior we observe depending on the order that packages are updated. By coupling update packages together, these problems will no longer have to be addressed.

But as system administrators, we need to be able to deal with these issues now without waiting for package development practices to change. We must fully understand the problems that our unique systems may face through global analysis. Understanding the causes of potential problems and recognizing warning signs when updating our systems will only help make our machines and systems more stable. Even without changing the way packages are developed, we can help to protect ourselves from the headaches of validation drift by using proper practices. For example, installation order of packages must be carefully observed as it is up to the system administrator to detect problematic conditions and assure the packages are installed in the proper order.

Many of the practices mentioned above trade space for validation. Nowadays, space in systems is cheap, while validation is expensive. By changing the practices we use in administering Linux systems, particularly with RPMs, we can save ourselves time and effort. The cost of this is the added space that multiple copies of libraries and packages may take up. But when you weigh the benefits versus the drawbacks, it is clear that a change in practices will help everyone.

Future Work

While the global analysis techniques upon which we report are *based* upon the strategies in sowhat, we have yet to integrate these into the tool proper. While we have a good grasp of the problem, a truly practical methodology seems to require this integration. We expect to do this some time in the coming year.

Further, we intend to look at the arbitrary script actions performed by RPMs before and after installation. These scripts can cause numerous things in a system to change; things that the user probably does not know are being changed. By looking at these scripts and comparing them against one another, we will be able to get an even better handle on exactly what an RPM is doing when updating a system in a baseline state.

But this work is just the tip of the iceberg. Homogeneity of packages is *necessary*, but not

sufficient. A truly practical strategy would account for all dependencies; not just those one can discover with `ldd`, but also dependencies that can only be discovered by tracing library references. We and Yizhan Sun are also working on wrapping library calls to trace perhaps non-conventional use of dynamic libraries (using `dlopen`), and even tracing – at a fine grain – actual file use in a live system. These measures will give us a better idea of the true dependencies in a running system that can be violated by poor practice.

Open Questions and Controversies

This work also brings up some rather important open questions for study by the whole community of RPM users. These are not questions that we feel that we can address ourselves with the technology we have. We leave them to other researchers.

One of the most heated controversies in current systems management is whether binary equivalence is necessary for behavioral equivalence of programs, libraries, or systems [10, 11]. It seems that the community is strongly divided into factions of “theorists” and “practitioners.” Some “practitioners” believe that only identical binary files are guaranteed to behave identically (our premise) and that differing compilers, for example, cannot be trusted to compile the same source code with identical behavioral results. Some “theorists” believe, however, that “we should be able to write compilers that perfect” and that the source code should be the real measure of equivalence. Some extremists also argue that even differing source code can be proved behaviorally equivalent by compiler optimization techniques. We take the very conservative position that “only practical techniques can be applied now” and thus believe binary equivalence the only currently practical measure of behavioral equivalence. Only time will tell whether the other ideas of equivalence will be practical.

Another open question concerns the general nature of dependency. So far, we can only describe dependencies in a very coarse way, by saying which files should be present or which packages should be installed. Dependency, in general, is a much more complex thing. It creates limits on the *contents* of files as well as their presence and location. How far can we go with describing dependencies before the cure (of discovering and declaring dependencies) is worse than the disease (of dependency failure)?

Conclusions

We have shown that problems in RedHat installations are *not* always caused by problems with dependencies between packages, but instead (and perhaps more commonly) by *overlaps* between packages. Dependencies declared inside a typical collection of RPMs are surprisingly accurate. But overlaps between package files seem to be a plague upon both closed and open repositories containing reusable binary RPMs.

In casually installing RPMs in a Linux system, it is easily possible to put the system into a state that no one has validated or tested. While for a “home computer” the risk of down time is fairly low, in an enterprise management strategy, such ad-hoc system updates should be avoided in favor of staying near configurations that have been extensively tested and “burned in” by the community. We show that deviations from tested states can sometimes be detected before an RPM is installed by global analysis of all RPMs available. We also suggest that RPMs be constructed so that any combination, in any order, always results in a validated system state. This is easy to accomplish by isolating dependencies and avoiding inhomogeneous overlaps, but seemingly only the major distributions have managed to do this properly.

Acknowledgements

Our advisor Alva Couch helped us with the scripts, resources, and writing involved in completing this paper. We would also like to recognize the ongoing and much appreciated efforts of RedHat, RpmFind, and other organizations dedicated to making software package installation more efficient, predictable, and robust. We do not intend anything in this article as a criticism of their noble efforts. In this paper we have shown how hard their job really is.

Author Biographies

Jeffrey D'Amelia is a graduate student at Tufts University working towards his Masters degree which will be completed in the Spring of 2003. Beginning in January 2002, Jeff was awarded a fellowship in the NSF GK-12 program. Through this program, he works at a junior high school in Malden, MA with the goal of infusing computer science problem solving approaches into the K-12 math curriculum. He has also worked at the college level as both an undergraduate and graduate teaching assistant for several different courses. Jeff can be reached via U. S. Mail at Tufts University, Department of Computer Science, Halligan Hall, 161 College Ave., Medford, MA 02155 or electronically at jdamelia@eecs.tufts.edu

John Hart is a graduate student at Tufts University working towards his Masters degree which will be completed in the Spring of 2003. He received his Bachelor of Engineering degree from Tufts University in Computer Engineering and has worked as undergraduate and graduate teaching assistant. John can be reached via U. S. Mail at Tufts University, Department of Computer Science, Halligan Hall, 161 College Ave., Medford, MA 02155 or electronically at jhart@eecs.tufts.edu

References

- [1] Bailey, Ed, “Maximum RPM,” SAMS, Inc., 1997.
- [2] Bodnar, Ladislav, “Is RPM Doomed?”, <http://www.distrowatch.com/article-rpm.php>.

- [3] Brooks, Fredrick, "The Mythical Man-Month," Addison-Wesley, Inc, 1995.
- [4] Couch, Alva and Yizhan Sun, "Global Analysis of Dynamic Library Dependencies," *Proceedings LISA 2001*, San Diego, CA, 2001.
- [5] Couch, Alva, "The Maelstrom: Network Service Debugging via 'Ineffective Procedures,'" *Proceedings LISA 2001*, San Diego, CA, 2001.
- [6] Couch, Alva, "SLINK: Effective Abstractions for Community-Based Administration," *Proceedings LISA 96*, San Diego, CA, 1996.
- [7] Hess, Joey, "A comparison of the deb, rpm, tgz, slp, and pkg package formats," <http://www.kitenet.net/~joey/pkg-comp/>.
- [8] Levine, John, "Linkers and Loaders," First Edition, Morgan Kaufmann Publishers, 1999.
- [9] Pressman, Roger, "Software Engineering: A Practitioners' Approach," Fifth Edition, McGraw-Hill, Inc, 2001.
- [10] Traugott, Steve and Joel Huddleston, "Bootstrapping an Infrastructure," *Proceedings LISA XII*, USENIX Association, 1998.
- [11] Traugott, Steve, and Lance Brown, "Why Order Matters: Turing Equivalence in Automated Systems Administration," *Proceedings LISA XVI*, USENIX Association, 2002.
- [12] Watkins, John, "Testing IT: an Off-The-Shelf Software Testing Process," Cambridge University Press, 2001.
- [13] "Debian GNU/Linux," <http://www.debian.org>.
- [14] The Linux Standard Base, "The Linux Standard Base Project," <http://www.linuxbase.org>.
- [15] The Slackware Linux Project, <http://www.slackware.com>.

RTG: A Scalable SNMP Statistics Architecture for Service Providers

Robert Beverly – MIT Laboratory for Computer Science¹

ABSTRACT

SNMP is the standard protocol used to manage IP networks. Service providers often analyze the utilization statistics available from SNMP-enabled devices to make informed engineering decisions, diagnose faults and perform billing. However collecting and efficiently storing large amounts of time-series data quickly, without impacting network or device performance, is challenging in very large installations. We identify three crucial requirements for an SNMP statistical solution: (i) support for hundreds of devices each with thousands of objects; (ii) the ability to retain the data indefinitely; and (iii) an abstract interface to the data. We then compare the applicability of several tools in a service provider environment. Finally, we detail Real Traffic Grabber (RTG), an application currently in use on our national IP backbone which we developed in lieu of existing packages to meet our requirements.

Introduction

Data traffic statistics are valuable in all networks, but are particularly crucial in service provider and enterprise environments. Not only is the data used to make informed engineering decisions such as traffic engineering, capacity planning and over-subscription analysis, it is also used for denial-of-service tracking, billing and policy purposes. The Simple Network Management Protocol (SNMP) [4] is the standard protocol used for fault detection, diagnostics, device management and statistics gathering in IP networks. While the protocol mechanisms themselves are “simple,” the process of continually collecting, retaining, reporting and visualizing SNMP statistics data presents unique constraints in very large network installations.

Worldcom is a large service provider with many disparate data networks and equally varied management systems. The particular Worldcom network we monitor is a national OC-48c (2.5 Gbps) backbone that has grown to approximately 110 devices each with an average of 100 interfaces. Scalability problems regularly plague service providers and other large networks scrambling to keep up with growth. The legacy systems gathering SNMP interface utilization statistics from our network were no exception and faced severe performance problems to the extent that they were unusable. Simultaneously, new requirements to monitor additional per-interface statistics emerged along with a need to generate various custom reports.

At a minimum, we needed a new system that could record bytes, packets and errors for every interface in the network with a five-minute granularity. The system must also produce long-term (multiple-year) trends and reports and keep detailed usage information for billing and legal purposes. We identified three high-level requirements for our new system: the ability to (i)

scale the statistics infrastructure to support hundreds of devices each with thousands of objects; (ii) retain the data indefinitely; and (iii) provide an abstract interface to the data. These requirements motivated the development of Real Traffic Grabber (RTG).

RTG is a flexible, scalable, high-performance SNMP statistics monitoring system. All collected data is inserted into a relational database that provides a common interface for applications to generate complex queries and reports. RTG has many unique properties including: it runs as a daemon incurring no **cron** or kernel startup overhead, it is written entirely in C for speed incurring no interpreter overhead, it is fully multi-threaded for asynchronous polling and database insertion, it performs no data averaging and it can poll at sub-one-minute intervals. RTG runs in production on several networks and has proved to be an invaluable tool.

In this paper we first compare the applicability of various open-source tools and solutions in a service provider statistics environment. Next we detail the implementation and operation of RTG. We then present performance data measured for various monitoring platforms including RTG. Finally, we present graphs and reports unique to RTG. The paper concludes with availability information and suggestions for future development.

Survey of Existing Monitoring Applications

There are many open-source tools for gathering SNMP data; CAIDA² maintains an excellent list of Internet measurement tools [3]. We experimented with several of the most popular applications and while they were ideal for many circumstances, none fulfilled our complete requirements.

A widely popular open-source application for visualizing link traffic is the Multi Router Traffic

¹The initial research was completed while with Worldcom.

²Cooperative Association for Internet Data Analysis, <http://www.caida.org>

Grapher (MRTG) [7, 8]. MRTG is a Perl script that reads a configuration file and SNMP polls the listed devices. An external C program adds the result to an ASCII log file and then produces a series of traffic plots and corresponding HTML code. MRTG assumes that as the data ages, the importance of detailed information diminishes proportionately. Subsequently, MRTG implements a lossy storage mechanism whereby multiple older samples are averaged into a single data point representative of the entire time period to ensure a fixed-size database.

The primary advantages of MRTG are its ease of setup and use and friendly web output. While the visualization piece was excellent, MRTG was not suitable for our environment for several reasons. With such a large network, MRTG could not poll and process all of the objects in the network within a five-minute interval. A MRTG process that did not complete on time led to multiple MRTG processes piling up, as MRTG is forcibly invoked via `cron` each sampling interval, exacerbating the speed problem further. The MRTG performance problems are due to its use of a flat ASCII log file, sequential SNMP polling and insistence on generating a new graphic image (either GIF or PNG) for each object every five-minutes. We examine the performance characteristics of MRTG in detail in a subsequent section. A second disadvantage with MRTG lies in its use of a fixed size database which guarantees decreasing resolution with time and the eventual discard of old samples. Further, the ASCII log file is difficult for other applications to interface with or correlate to a particular set of customers easily.

In response to the performance issues in MRTG, the primary MRTG author created the Round Robin Database tool (RRDtool) [6] which re-implements the lossy storage technique found in MRTG in a pure binary format for speed improvements. RRDtool also offers much more flexibility than MRTG, allowing multiple data sources per archive, varying time resolutions and non-integer values. The graphing facilities are similarly flexible, now allowing on-demand graphs for arbitrary time periods and the ability to draw multiple

data sources simultaneously. The redesign, however, does not address the data gathering (SNMP polling) aspect of the performance problem and does not meet our requirement to retain all data samples indefinitely.

Cricket [1] is designed to provide a manageable interface to RRDtool; its hierarchical configuration tree using inheritance works particularly well for large networks. Cricket is a set of Perl scripts that gather information about the network topology, set up RRDtool properly, and use the Perl SNMP module [5] to collect statistics. The end result is a set of easy to navigate web pages with RRDtool traffic plots similar to those provided by MRTG. We were impressed by the ease of configuration and useful web output of Cricket. While Cricket combined with RRDtool offers impressive flexibility and speed, we desired a more generic interface to the data and felt that there were more speed improvements to be had. Cricket with RRDtool still incurs `cron` and Perl interpreter overhead, sequentially polls devices and averages data samples.

Table 1 summarizes the advantages and disadvantages of the open-source tools we evaluated and the primary use of each. We include RTG in this table for comparison. While MRTG, RRDtool and Cricket are appropriate for different environments, none met our performance or collection criteria. Schemes that perform long-term averaging can hide link peculiarities important to engineering. Based on the availability and relative low cost of fixed disk storage, our design constraint was to keep long-term data indefinitely without averaging. We also recognized that it was impossible to anticipate every user or application that would need access to the data. Thus, we wanted as abstract and open of an interface to the data as possible. The creation of RTG was motivated by the lack of suitable open-source tools and the inflexibility of available commercial solutions.

RTG Implementation

From our base requirements as a large service provider and our experience with other SNMP statistics

Tool	Advantages	Disadvantages	Primary Use
MRTG	Ease of setup and maintenance, friendly web output, large user base	Performance problems for large networks, lacks flexibility and external interfaces	Small networks requiring only traffic plots
Cricket with RRDtool	Highly configurable, web output, high performance, large user base	Averages samples, incurs Perl and <code>cron</code> overhead	Mid-to-large networks
RTG	Very-high performance with asynchronous threaded polling, uses SQL database for applications to generate complex queries and reports, supports sub-one-minute polling intervals, runs as a daemon	Requires external packages (Net-SNMP and MySQL), complex to configure, steep learning curve	Large-to-very large networks that require traffic plots, advanced reports, no data averaging and indefinite data storage

Table 1: Comparison of select SNMP statistics monitoring tools.

packages, we developed RTG. RTG is comprised of **rtgpoll** (a polling daemon), **rtgplot** (a plotting program), **rtgtargmkr.pl** (a network configuration parser), a collection of Perl reporting scripts and a set of PHP scripts to provide a web interface.

The RTG system centers around the polling daemon, **rtgpoll**. To provide the highest performance possible, the poller is written in C and runs as a daemon, utilizing less memory and fewer processor resources. Further, to allow asynchronous parallel querying and prevent any single query from blocking other polls, **rtgpoll** is fully multi-threaded. A thread per SNMP query is used such that the poller maintains a constant number of “queries in flight,” greatly improving performance.

An often overlooked performance problem lies in the network devices themselves. The SNMP agent on many IP devices consumes significant resources in response to queries, particularly when many objects are polled. To equalize the query load and prevent device CPU starvation which may inadvertently cause routing problems or service instability, RTG randomizes the target list before polling. In this manner, even if the target file lists devices sequentially, **rtgpoll** SNMP queries individual object identifiers (OIDs) of the devices at random. Whereas our previous SNMP management software inflicted noticeable CPU spikes on the network devices, no spikes are evident with this scheme. An added benefit to this randomization strategy is that should a device be physically down or unreachable, all of the **rtgpoll** threads will not block waiting for the device query to timeout. Thus, a single device that is unreachable has little or no impact on the overall RTG poll cycle time.

The **rtgpoll** program reads a master configuration file, **rtg.conf**, and a target file. The configuration file contains general RTG parameters while the target file contains the list of SNMP targets, SNMP communities, OIDs, SQL tables and other information. An auxiliary Perl script included in the RTG distribution, **rtgtargmkr.pl**, maintains the target file and ensures that the interface information in the database is kept up to date. **rtgtargmkr** reads a list of devices, SNMP fetches each interface name, description and speed, and maintains database consistency. For instance, the SNMP interface identifier for a particular interface can change between network device reboots or after adding a new interface to a network element. The **rtgtargmkr** script manages these changes and detects new interfaces or interface description changes. The target list is re-read when **rtgpoll** traps the UNIX HUP signal, allowing for dynamic reconfiguration without restarting the daemon. We run **rtgtargmkr** periodically via a **cron** job and then send **rtgpoll** a HUP signal so that RTG always maintains an accurate view of the network.

Each target is SNMP polled at the interval specified in the **rtg.conf** file and the result is inserted into a MySQL [11] database. We chose MySQL because it is open source, very fast, portable, has a large installed user base and has multiple application programming interfaces (APIs), including C, PHP, Perl. While it is technically feasible to use other databases, for instance for users with existing database infrastructure, this would require significant reprogramming of RTG. We are very pleased with the performance and stability of MySQL for RTG and have seen little interest in using other databases. Figure 1 presents a functional diagram of the RTG system.

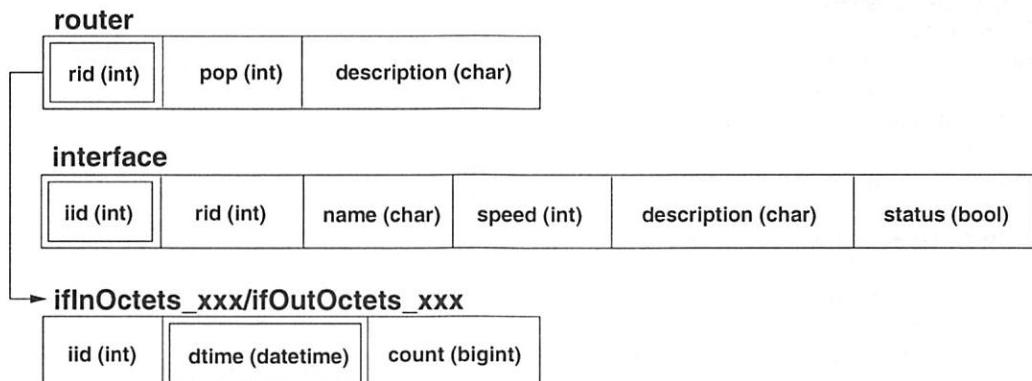


Figure 2: RTG database schema.

```

mysql> SELECT * FROM ifInOctets_9 WHERE iid=117
      AND dtime>'2002-07-14 18:50' ORDER BY dtime LIMIT 3;
+-----+-----+-----+
| iid | dtime           | counter |
+-----+-----+-----+
| 117 | 2002-07-14 18:51:16 | 5092874 |
| 117 | 2002-07-14 18:56:23 | 5857165 |
| 117 | 2002-07-14 19:01:25 | 4762324 |
+-----+-----+-----+

```

Listing 1: MySQL query illustrating use of the RTG schema.

RTG can poll either 32 or 64-bit integers and gracefully handles counter wrap and anomalous values. Counter wraps are detected when the SNMP result is less than the previous SNMP result from the last sample interval for a particular OID. When RTG encounters a counter wrap, the database insert value is calculated as either

$$(2^{32} - \text{last_value}) + \text{current_value}$$

or

$$(2^{64} - \text{last_value}) + \text{current_value}$$

depending on the OID integer size. Unfortunately, in practice some counter wraps are not legitimate. Often if a device is rebooted between polling intervals, the SNMP value returned after the reboot will be less than the previous value RTG maintains. RTG eliminates these bogus data points by defining a configurable out-of-range value above which `rtgpoll` will never attempt an insert into the database. The out-of-range value is typically configured as a multiple of the maximum number of bytes possible in the defined interval on the highest speed link.

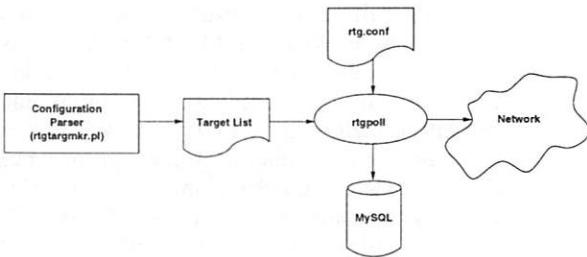


Figure 1: RTG system functional diagram.

To meet the long-term storage requirement, RTG utilizes the MySQL database in combination with a highly efficient database schema. Every effort was made to minimize the amount of data that must be stored and maximize performance. We observed that most reporting and analysis applications are interested in router or interface specific data for a particular object, such as byte counts, over a time range. In order to minimize the amount of data any single query would have to process, minimize the amount of data stored and to segment the data as much as possible, we created a SQL table per unique device and object. Each table name contains the device identifier (rid), i.e., `ifInOctets_9`. In this unconventional fashion, the table name becomes significant as a unique index. Each of these tables contains only interface identification (iid), date/time and count columns. The table is further indexed by the date/time (dtm) column. Two additional tables provide router and interface index identifiers (rid and iid) as well as descriptions and names. The database schema is illustrated in Figure 2.

A potential drawback to this schema is that each device requires five tables corresponding to five or more files on the MySQL system. Because of this, the maximum number of devices is bounded by the operating system and MySQL's ability to speedily handle

many files. Despite this limitation, this method has allowed us to retain more than two-years of complete data for over 100 devices without performance impact; reports for old data are generated as quickly as reports for new data. While different schemas may be more appropriate for other installations, this database schema provides the highest performance in our network. It is important to note that RTG does not impose any requirement to use this schema. In fact, the RTG table names are completely configurable in the target list file. For instance, some installations choose to use only five tables total corresponding to input and output octets, packets and errors.

Using the aforementioned schema for each unique network element, the interface identifier (iid), timestamp and difference between the last SNMP sample and the current poll are inserted into the network element's unique table. Assume that a user has identified a device and interface of interest based on the device and interface descriptions in the RTG database. If the device and interface in question are `rtr1.someplace` with a router identifier (rid) of 9 and interface id (iid) 117 respectively, Listing 1 shows the MySQL query (limited to the first three rows) to gather `ifInOctet` data.

Thus, only the absolute minimum amount of data is stored in the database preserving speed and storage space. On one production MySQL server, RTG is using approximately 5.5 GB of data and 3.9 GB of index space (total of 9.4 GB) to store two-years of data.

We do not enforce any data periodicity in the database; it is the responsibility of the application to determine the total time elapsed between subsequent samples for any given table and interface should the application need to calculate traffic rates. RTG does not record rates, only absolute counts. In the previous example query, an application would calculate the rate as

$4,762,324 \text{ Octets}/302 \text{ sec} = 15.8 \text{ KBps} = 126.2 \text{ Kbps}$. Finally, RTG's high-performance has the added advantage of allowing sub-minute polling intervals for instances where high sample granularity is required. We present a real-world example of the utility of sub-minute polling in the RTG Reports section.

Performance Evaluation

Because performance is a central component of RTG, we evaluated RTG, MRTG and Cricket for speed. All tests were performed on a dual 360 MHz Sun Ultra 60 workstation running the Solaris 2.7 operating system. We used the UNIX time command to observe the total execution time, user CPU and system CPU times for each application. We measured the performance five times and then took the simple mean of the five test runs. Each application's CPU utilization is presented in Table 2. While MRTG and Cricket use significantly more processor cycles than RTG, this data does not

include the CPU utilization of RTG's MySQL server. Despite this, MySQL RTG CPU usage has never been a practical limitation in production environments and we note that it is possible to run the RTG polling daemon and the MySQL database on separate physical machines if needed.

The application performance data is presented in Table 3. Because in normal operation RTG runs continuously as a daemon, we modified the code to exit after five polling cycles thereby allowing us to use the UNIX time utility. The data shows that Cricket with RRDtool is far superior to MRTG. Cricket and a non-threaded version of RTG are comparable in speed, although RTG uses fewer CPU resources. Finally, the multi-threaded version of RTG is by far the fastest application in the group achieving approximately 107 targets per second in our testing. Assuming the traditional five-minute sample interval, this yields a theoretical maximum of 32,000 OIDs monitored on a single RTG system before saturation, almost five times as many as Cricket and twenty-four times more than MRTG.

Whereas with other systems it is not possible to query just byte statistics on the entire network within the sample period, the speed of RTG allows us to not only monitor all devices in the network, but also to monitor additional objects per interface. For example, we now monitor the SONET Management Information Base

(MIB) [10] to proactively track transmission problems and the MPLS MIB [9] to analyze Label Switched Path (LSP) traffic. Every five minutes, our production RTG system processes approximately 5000 OIDs in approximately 60 seconds leaving ample room for future growth. We suspect that even higher performance is possible by increasing the number of threads, and hence the number of queries in flight, beyond the default of five. Because the performance is more than acceptable, we are hesitant to increase the number of threads on our production RTG for fear of overwhelming the network or the network devices. We note also that the architecture of RTG easily allows separation of the various components. For instance, multiple instances of the RTG poller could be distributed throughout the network while utilizing separate physical machines for MySQL and web page or report generation to achieve even greater scalability.

RTG Reports

Our experience shows that using a SQL database provides an ideal abstract interface to the data. The available Perl, PHP or C API's facilitate rapid prototyping and allow for the design of highly customized reports and tools. In our case, the engineering group currently receives a nightly traffic report including total bytes, packets, maximum rate and average rate for the backbone routers via a scheduled Perl DBI script. Each

Application	Targets	User CPU (s)	System CPU (s)	CPU %
MRTG	1618	113.8	19.0	36.1
Cricket	2010	21.3	0.7	25.1
RTG	2255	1.3	0.7	0.6
RTG-threads	3650	1.7	1.1	0.8

Table 2: Application CPU utilization.

Application	Target's	Run Time (s)	Sec/Target	Targets/Sec	Max Targets in 5 min
MRTG	1618	365.4	0.23	4.43	1328
Cricket	2010	87.8	0.04	22.89	6868
RTG	2255	77.6	0.03	29.06	8717
RTG-threads	3650	34.2	0.01	106.73	32018

Table 3: Application performance.

Traffic Daily Summary							
Period: [01/01/1979 00:00 to 01/01/1979 23:59]							
Site	GBytes In	GBytes Out	MaxIn(Mbps)	MaxOut	AvgIn	AvgOut	
<hr/>							
rtr1.someplace:							
so-5/0/0	384.734	360.857	49.013	43.420	35.630	33.426	
so-6/0/0	357.781	421.736	42.923	50.861	33.137	39.053	
tl-1/0/0	0.054	0.058	0.005	0.006	0.005	0.005	
rtr3.someplace:							
so-6/0/0	1,115.258	1,246.163	168.776	172.690	103.173	115.439	
so-3/0/0	1,142.903	1,028.256	152.232	162.402	105.863	95.142	
so-7/0/0	152.824	199.742	22.052	35.005	14.152	18.488	

Listing 2: RTG summary traffic report.

morning engineers can peruse this email for interesting events that may require attention. Listing 2 shows example output from the Perl traffic summary report

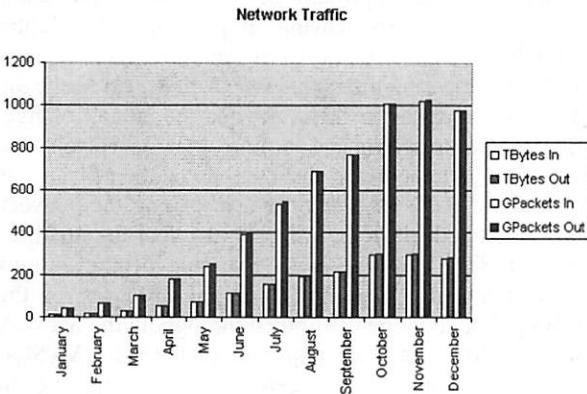


Figure 3: Example year-to-date traffic plot from RTG generated CSV data.

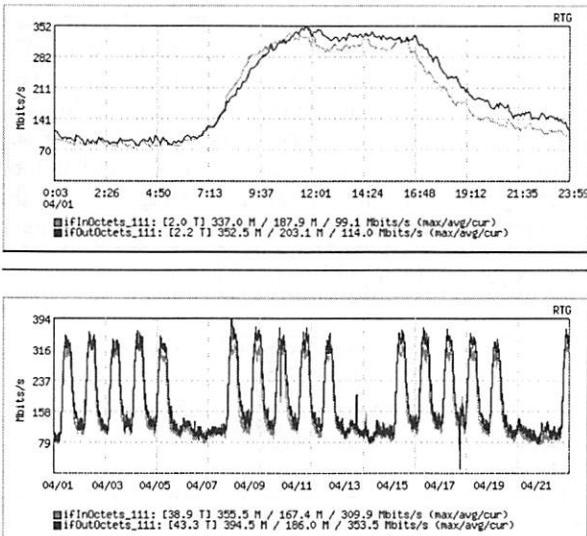


Figure 4: RTG plots for different time scales with no loss of resolution.

included in the RTG distribution. Because no averaging is used, absolute numbers such as the total number of Gigabytes are shown and the report result for a specific time period will be the same regardless of when the report is generated. This consistency is invaluable for accurate trending and accountability.

Because the data is stored in a relational database, it is straightforward to generate a traffic report for a

single customer or a subset of customers for any arbitrary time period. Listing 3 depicts output from the 95th percentile Perl report included with the RTG distribution. This report shows a customer's usage on three circuits including their 95th percentile rate, a metric often used for billing in the telecommunications industry.

Another Perl script we use regularly generates year-to-date trunk utilization data in comma separated value (CSV) format, a format that is easily imported into commercial spreadsheets. These reports are used by capacity planning groups and presented to upper management. An example graphic of year-to-date traffic generated by plotting the CSV data in a spreadsheet is shown in Figure 3.

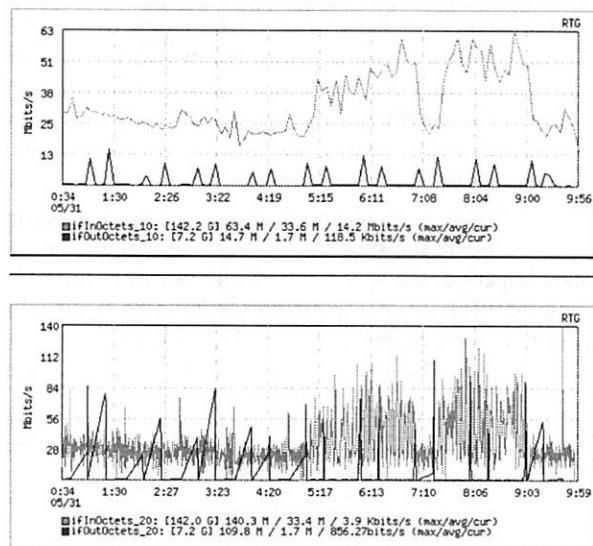


Figure 5: Effect of differing sampling rates measuring the same circuit and time period using 5 minute sampling (top) and 30 second sampling (bottom).

RTG includes a set of PHP web pages that provide a graphical view of circuit utilization for customers and support staff for any arbitrary time period. A key application included in the RTG distribution for web pages and traffic visualization is **rtgplot**. **rtgplot** is a C program that utilizes the GD library [2] to generate plots, similar to those generated by MRTG, in PNG format from the RTG database. **rtgplot** provides an extremely fast on-demand graphical interface to the data. **rtgplot** can be used as a stand alone application or embedded in web pages. A traffic plot can be placed

ABC Industries Traffic
Period: [01/01/1979 00:00 to 01/31/1979 23:59]

Connection	RateIn Mbps	RateOut Mbps	MaxIn Mbps	MaxOut Mbps	95% In Mbps	95% Out Mbps
at-1/2/0.111 rtr-1.chi	0.09	0.07	0.65	0.22	0.22	0.13
at-1/2/0.113 rtr-1.dca	0.23	0.19	1.66	1.12	0.89	0.57
at-3/2/0.110 rtr-2.bos	0.11	0.16	0.34	0.56	0.26	0.40

Listing 3: Customer 95th percentile traffic report.

easily in any web page by simply using the HTML tag with the appropriate arguments. For instance

```
<IMG SRC="rtgplot.cgi?t1=ifInOctets_2&t2=ifOutOctets_2&iid=49&begin=1028606400&end=1028692800&units=bits/s&factor=8&scalex=yes">
```

will plot two lines of traffic data from the RTG MySQL tables ifInOctets_2 and ifOutOctets_2 corresponding to interface id 49 on router id 2 for the 24 hour period 1028606400 to 1028692800 (UNIX seconds since the epoch). The ‘factor’ argument multiplies the byte data to produce bits per second output on the plot, while the ‘units’ argument will be displayed on the vertical axis. The ‘scalex’ argument will auto adjust the horizontal time axis according to the available data samples rather than according to the actual time span specified. Non-continuous data, such as errors, are plotted using the ‘impulses=yes’ argument. Different time views, with no resolution decay, of the same circuit monitored by RTG are shown with **rtgplot** output in Figure 4. Note that each plot includes the absolute volume of bytes as well as the maximum, average and current traffic rates for the time range.

Finally, the utility of an SNMP tool that is extremely fast and supports sub-one minute polling is underscored by a recent example where a customer OC-3c circuit (155 Mbps) was experiencing performance degradation due to packet loss. Quickly examining the RTG plot for the circuit did not immediately reveal any congestion. We then configured RTG to poll this interface every 30 seconds rather than every five-minutes. Figure 5 shows two plots of this interface over the same time period. The upper plot is the result from polling every five minutes whereas the lower plot is the result from polling every 30 seconds. Clearly the five-minute polling interval masked the customer’s traffic bursts that were causing packet loss. While the plot generated from five-minute samples shows a peak input rate of 63.4 Mbps, the plot generated from the 30-second samples shows a peak input rate of 140.3 Mbps. Data averaging would mask this type of problem even further, particularly as the data aged.

Future RTG Development

We are continuing to develop RTG and improve it based on feedback from the open-source community. In particular we are looking to increase the robustness of RTG by employing a buffering mechanism to buffer SNMP results in case the SQL database is down or unreachable. In addition, we want to develop functionality by which multiple RTG clients can communicate with one another to provide distributed polling and redundancy. We recognize that setup and installation of RTG is difficult and we plan to improve the configuration utilities, documentation, etc. Finally, we have received feedback about additional uses of RTG including implementing multi-grain storage techniques, as opposed to the traditional fixed sample interval, to

isolate interesting variations in the data. This could potentially lead to the development of a denial-of-service detection or fault management system.

RTG Availability

RTG is developed on Solaris, tested on FreeBSD and Linux, and should run on a wide variety of other UNIX platforms by virtue of a GNU **autoconf** script. There is no support for Windows platforms. RTG is available under the terms of the GNU GPL from the RTG web page hosted on SourceForge, <http://rtg.sourceforge.net>. Further information, including documentation, and mailing lists can be found on the RTG home page.

Acknowledgments

RTG was inspired by the excellent tools from Tobias Oetiker, Dave Rand and Jeff Allen. RTG uses, and is useless without, the MySQL, UCD SNMP, gd, png and cgilib packages. The author would also like to thank Kevin Thompson for his support of this work.

Author Biography

Robert Beverly is currently pursing a PhD in Computer Science at the Massachusetts Institute of Technology. Most recently he was a senior engineer with Worldcom’s Advanced Internet Technology group in Northern Virginia where he was responsible for the statistics and measurement infrastructure of several large networks. Prior to Worldcom’s acquisition, Mr. Beverly worked for MCI Internet Engineering on the very-high-performance Backbone Network Service (vBNS). He received his Bachelor’s degree in Computer Engineering from the Georgia Institute of Technology in 1996 with high honors. While at Georgia Tech, he worked for several years managing the campus UNIX systems and then spent two years working for the Office of Information Technology managing the campus backbone network. Reach the author at rbeverly@mit.edu.

References

- [1] Allen, J., “Driving by the rear-view mirror: Managing a network with cricket, *Proceedings of the First Conference on Network Administration*, April 1999.
- [2] Boutell, T., *GD graphics library*, <http://www.boutell.com/gd>.
- [3] CAIDA, *Internet tools taxonomy*, <http://www.caida.org/tools/taxonomy/>.
- [4] Case, J. D., M. Fedor, M. Schoffstall, and C. Davin, *Simple Network Management Protocol (SNMP)*, RFC 1157, Internet Engineering Task Force, <ftp://ftp.ietf.org/rfc/rfc1157.txt>, May 1990.
- [5] Leinen, S., *Perl 5 SNMP module, an SNMP client implemented entirely in Perl*, <http://www.switch.ch/misc/leinen/snmp/perl>.

- [6] Oetiker, T., *RRDtool*, <http://people.ee.ethz.ch/~oetiker/webtools/rrdtool>.
- [7] Oetiker, T., "MRTG – The Multi Router Traffic Grapher," *Proceedings of LISA 1998*, December 1998.
- [8] Oetiker, T. and D. Rand, *MRTG*, <http://people.ee.ethz.ch/~oetiker/webtools/mrtg>.
- [9] Srinivasan, C., A. Viswanathan, and T. Nadeau, *Multiprotocol Label Switching (MPLS) Label Switch Router (LSR) management information base*, Internet-draft, Internet Engineering Task Force, <ftp://ftp.ietf.org/internet-drafts/draft-ietf-mpls-lsr-mib-08.txt>, January 2002.
- [10] Tesink, K., *Definitions of Managed Objects for the SONET/SDH Interface Type*, RFC 2558, Internet Engineering Task Force, <ftp://ftp.ietf.org/rfc/rfc2558.txt>, March 1999.
- [11] Widenius, M., D. Axmark, and A. Larsson, *MySQL AB*, <http://www.mysql.com>.

Environmental Acquisition in Network Management

Mark Logan, Matthias Felleisen, and David Blank-Edelman

– Northeastern University

ABSTRACT

Maintaining configurations in heterogeneous networks poses complex problems. We observe that medium and large networks exhibit many contextual relationships, and argue that modeling these relationships explicitly simplifies configuration management. This paper presents a declarative data specification language, called Anomaly, that implements our ideas. Anomaly models containment relationships and uses a data aggregation technique called environmental acquisition to simplify system management. The interpreter for the language generates and deploys configurations from a source code description of the network and its hosts.

Introduction

Configuration management is an important task for system administrators, as it is often one of the largest portions of their job. In the last decade, a number of configuration management systems have emerged to help with this task. Some of this work is summarized in a paper by Remy Evard [1].

Our paper presents an attempt to take some of the best features of previous solutions and use them in a framework based on *environmental acquisition*, an abstraction mechanism borrowed from the object-oriented systems community. The result is a declarative language, called Anomaly. Anomaly is also intended to be useful in practice as well as in theory. It has a plug-in interface for adding extensions.

Previous Solutions

A survey of previous solutions to the problem of generating and managing configurations reveals three particularly important concepts: data aggregation techniques such as inheritance and class systems; logic programming techniques that reduce the complexity of configuration statements; and database-driven systems that store configuration data in a repository.

Cfengine [4] is one widely used solution. It is a language-based host configuration tool that uses a class system to aggregate configuration commands. Cfengine's class system allows an administrator to apply configuration statements to a class of machines. It uses techniques similar to logic programming to make its statements more concise. Couch and Gilfix demonstrated this in their 1999 paper, “It’s Elementary Dear Watson...” [5]. Cfengine is not perfect, however. Its host-centricity does not lend itself well to other components of the environment, especially switches and routers.

Language-based configuration tools are an important contribution, but for large networks, configuration data can become unwieldy when stored in

source code form. Database-driven approaches [2, 3] solve this problem by providing a repository for configuration data. This approach simplifies the maintenance of configurations and reduces the rate of errors by reducing the amount of source code that administrators have to deal with.

The paper by Couch, et al., [5] also presents an approach that leverages convergent processes based on logic programming (Prolog). The authors point out that previous approaches, especially Cfengine, already use convergent processes. An example of this is the Cfengine link command, which looks like:

```
links:  
/etc/sendmail.cf ->! mail/sendmail.cf
```

In Cfengine, the link command (like most other commands) hides a great deal of housekeeping from the administrator. The link command above takes care of checking whether /etc/sendmail.cf already exists as a link to mail/sendmail.cf, or if it already exists as a non-link file, and takes appropriate actions to make /etc/sendmail.cf a link to mail/sendmail.cf. To do the same in Bourne shell might take a dozen lines.

This process is called a convergent process because executing the link statement multiple times does not have side-effects after the first execution (i.e., it is idempotent). Also, the link statement behaves appropriately regardless of the initial state of the system. Therefore, the state of the system *converges* to the ideal state described by the source.

Environmental Acquisition

The fundamental insight of this paper is that all network objects exist in *contexts*. The idea of context dependency is both powerful and pervasive. Every computer, every switch, every printer in a network exists in a context that affects its desired behavior. In other words, the physical and logical location of a network object determines properties of that object. Here are some concrete examples of network objects and their context dependencies:

- **Access Controls.** A host in a computing lab must have less restrictive access controls than a host in a data center. Here, the physical location of the host is part of its context.
- **IP Configuration.** The netmask and default route of a host depend on the subnet to which the host belongs. Here, the subnet is the context of the host.
- **Switch Configuration.** Individual ports on network switches often require their VLAN membership to be set by an administrator. In networks where each subnet uses a separate VLAN, a switch port may depend on its context, i.e., the subnet to which it belongs, to determine its VLAN membership.
- **Printer selection.** Each computer lab or office in a building may have its own printer. This printer then becomes part of the context of each host in the room. Put simply, the most sensible thing for a host to do by default is to use a printer that has been assigned to the room in which the host is located. This can be accomplished by looking for a default printer in the physical context of each host.

System administrators use this contextual information almost every time they configure a network element such as a host, switch, or printer. To our knowledge, no tool exists to model this contextual information and automatically generate configurations based on it.¹ Therefore, the goal of Anomaly is twofold. First, Anomaly should encourage administrators to think actively about the contextual relationships in their network. Second, Anomaly should be a tool for modeling these relationships explicitly, in order to simplify the maintenance of configurations.

One method of modeling contextual information is to treat contexts as containers and construct a set of containment relationships regarding network objects. Using this idea, one notices relationships such as “the machine *chorf* is *in* room 201,” and “the machine *ambler* *belongs* to subnet 192.168.7.0/24.” From here, it is easy to observe that objects acquire properties from containers.

The name for this process is *environmental acquisition* [8]. Environmental acquisition, or simply acquisition, is an analogue of inheritance that operates on object/container relationships, rather than class/sub-class relationships.² The following example, paraphrased from the original paper on acquisition [8], illustrates the idea perfectly.

¹Cfengine does have a notion of context, in that it conditions its actions based on probes of the filesystem and operating system. However, it does not emphasize *modeling* contextual relationships, nor does it propose a single method of examining context, such as the construction of containment graphs.

²The Zope application server (<http://www.zope.org/>), the most prominent use of environmental acquisition, uses environmental acquisition to build and manage web content.

Consider a red car. If someone asks you about the color of the car’s hood, you will certainly tell them that the hood is red, unless you know otherwise. In this situation, the hood has *acquired* its color from the car. However, it would be wrong to model this relationship using inheritance, because a hood is not a type of car. A hood is a part of a car, that is, car and hood have an object/container relationship across which properties are acquired.

Through the use of acquisition, administrators can build models in which hosts acquire their access restrictions from their physical location, and their default routes from the subnet they reside in.

Many previous solutions use inheritance-like mechanisms for data aggregation. Although inheritance has worked well in previous approaches, it is not the most natural or useful data aggregation mechanism available.

First, the purpose of inheritance is not to aggregate data. At best, it can be considered a feature (or behavior) aggregation technique. Moreover, inheritance is used to establish *is-a* relationships. In the approaches discussed above, this rule is bent slightly (to great benefit, of course). For example, Cfengine’s class system uses boolean set operations (logical AND and OR) to decide to which machines a given action should be applied. The following statement says “link /etc/passwd-link to /etc/passwd on all machines that are in classes solaris and guest.”

```
solaris.guest:: # logical AND
    /etc/passwd-link -> /etc/passwd
```

This approach is loosely based around the idea of inheritance. Objects are instances of specific classes, and the class which an object belongs to determines its behavior. However, inheritance is not an appropriate technique for data aggregation in system management. Acquisition is a better approach to data aggregation for the following two reasons.

First, acquisition encourages system administrators to think about containment and contextual relationships. This is an improvement over inheritance, which encourages thinking in terms of *is-a* relationships. While it may be true that the host *chorf* *is-a* Solaris host, *chorf* also has interesting relationships with its surroundings. It is *in* a room, and it is *part-of* a subnet, but neither of these relationships lends itself to inheritance.

Second, acquisition has finer granularity than inheritance. Inheritance demands that a child class inherit all the features of its parent. A child class may choose to override certain features that it inherits, but it may not decline them outright. This is necessary because inheritance imposes sub-type relationships. A child class must have all the behavior of its parent, or else the *is-a* relationship is nullified. Because acquisition does not impose sub-typing, an object may pick and choose which

features to acquire from its container.³ This prevents objects from acquiring unexpected properties.

There is at least one other twist on inheritance that bears mentioning, namely *value inheritance*. Value inheritance is a data aggregation technique used in prototype systems such as ARK [6]. It works by making copies of a prototype object, and then making tweaks to the values inherited from the prototype. Most of the examples presented in this paper could also be implemented using value inheritance. We believe, however, that environmental acquisition is the better choice for the reasons discussed above. It encourages thinking in terms of contextual relationships, and it provides better granularity than value inheritance. David Ungar's paper on SELF [10] discusses prototyping and value inheritance in depth.

Logic Programming

Anomaly does not have the power of a full logic programming language such as Prolog. Cfengine's logic programming abilities also exceed that of Anomaly's. In Anomaly, the ability to specify the targets of configuration statements with 'facts' is replaced by the approach of context modeling and environmental acquisition. However, Anomaly keeps what we believe is the most important contribution of Cfengine, namely, the idea of convergent processes specified by logical statements. Therefore, rather than using imperative statements such as "add this user" or "put this interface in promiscuous mode," Anomaly uses logical assertions such as "this user must exist," or "this interface must be in promiscuous mode."

Anomaly

Our language, Anomaly, combines important elements of prior approaches with environmental acquisition. Anomaly has constructs to model containment relationships (i.e., which objects are contained in which objects), and constructs for describing the ideal state of individual objects.

Examples

The following examples illustrate the use of Anomaly and highlight its strengths, by examining several typical system administration scenarios. Several different types of contextual relationships are presented, in order to demonstrate the benefits of explicitly modeling context.

Host Access Controls

Configuration of host access controls is the most basic and obvious use of acquisition in system management. A system administrator can make access

³Acquisition comes in two flavors, implicit and explicit. When using implicit acquisition, any attempt to access a variable that is missing from an object results in an attempt to acquire that variable. When using explicit acquisition, no attempt is made to acquire variables unless they are listed explicitly as candidates for acquisition. Anomaly uses explicit acquisition.

controls more maintainable by exploiting a contextual relationship in the environment. In a university setting, it may be appropriate to use the physical location of a host as the context. In a corporate setting, the appropriate context may be the ownership of the machines (e.g., departmental, individual). In this example, we use the physical location as the context.

The first consideration is the containment graph. It is specified in Figure 1. Figure 2 defines some of these objects. The object CullinaneHall is defined as a Building, and is given a default access policy that allows *only* users in the systems netgroup to log in. This means that any host in the building acquires a restrictive access policy, unless otherwise specified.

```
CullinaneHall contains {
    UnixLab;
    DeansOffice;
}

UnixLab contains {
    chorf;
    wharf;
    staypuff;
}

DeansOffice contains {
    deans-laptop;
}

Building CullinaneHall {
    AccessPolicy access;
    access.allows(systems);
}

Room UnixLab {
    AccessPolicy access;
    access.allows(students);
}

SolarisHost chorf {
    AccessPolicy access('/etc/passwd')
    acquire access
}
```

Figure 1: Controlling access policies in object definitions.

For the UnixLab object, that is exactly what is done. Since the UnixLab object represents a public computer lab, it must have an access policy that allows users from the students netgroup to log in. Therefore, any host placed in the UnixLab object acquires a permissive access policy, specifically one that allows students to use the computers.

From this example, it is easy to see the benefits of using this method across an entire environment. When every room in a building has a sensible access policy assigned to it, administrators hardly have to worry about individual hosts. This directly addresses the recurring problem of hosts moving from faculty desks to public labs (Or from similar restricted access locations) If no one

remembers to edit `/etc/passwd`, the result is that students cannot log into the machine.

The most important benefit of this scheme is that even if a forgetful administrator moves a machine from one place to another without consideration for its access policy, the machine acquires a sensible access policy from its new environment. Modeling the context of the machine has thus made the path of least work more likely to be the *correct* path. In short, the lazy way produces the best defaults.

```

SwitchPort zaphod-8-13 {
    SwitchNum switchnum(switch =
        'zaphod.ccs.neu.edu');
    switchnum.is('8/13');

    acquire vlan;
    acquire ether;
}

SolarisHost chorf {
    MAC ether;
    ether.is('01:1C:ED:C0:FF:EE');
}

Subnet UnixSubnet {
    NetworkAddr net;
    Netmask mask;
    VLAN vlan;

    net.is('10.10.116');
    mask.is('255.255.254.0');
    vlan.isnamed('116');
}

UnixSubnet contains {
    chorf;
}

chorf contains {
    zaphod-8-13;
}

```

Figure 2: Modeling a switch Port.

Switch Port Configuration

Switch port configuration often involves configuring VLAN membership and port security settings. The contextual relationship is not quite as obvious as in the previous example. Here, it is necessary to treat the switch port as an object contained in the host that is attached to it. From a physical standpoint, this seems inappropriate. However, from a logical standpoint it is easier to think of the host being part of the port's context. When using MAC-based port security, it is necessary to know which host the switch port is supposed to serve.

Figure 2 shows the containment and object declarations for `chorf` and its switch port. When the configuration in Figure 2 is built, the switch port acquires two fields: `vlan` and `ether`. The `vlan` field allows the switch port to set its VLAN membership properly, and

the `ether` field allows it to set its port security settings properly. After the configuration is deployed, the switch will be usable only by `chorf`.

This figure also raises a question about the difference between the '`is`' assertion and the '`isnamed`' assertion. The former is used when an assertion describes the state of the variable. In this example, the `ether` field is completely specified by a 48 bit address, that is, the `ether` field *is* its address. The latter assertion, `isnamed`, is used when referring only to the identifier of a field. The `vlan` field in this example is only concerned with the identifier of the VLAN, not with configuration of that VLAN on the switch.

Beta-Test Environments

When upgrading subsystems such as daemons, kernels, or user applications, it is best to test the new software on a small subset of machines. It is possible to test upgrades on a single machine near the administrators' offices, but a single machine may not be representative of the rest of the environment (especially if host hardware is substantially varied throughout the environment). Furthermore, a designated test machine is not likely to have the same usage patterns as most machines in the environment. Anomaly can assist with this problem by providing a better method for organizing beta test systems, using environmental acquisition.

```

Platform Solaris {
    Packages packages;
    Patches patches;

    patches.has('sun-recommended');
    packages.has('openssh-3.0.2pl1');
    packages.has('lprng-3.6.14');
}

Platform Beta {
    Packages packages;
    Patches patches;

    patches.has('sun-recommended');
    patches.has('108604-18');
    packages.has('openssh-3.1pl1');
}

```

Figure 3: Platform objects.

The containment relationships described in Figures 3 and 4 simplify beta testing. In order to test a new Solaris patch (108604-18), we add it to the `Beta` container. This causes the patch to be installed on all machines contained in `Beta`. When the patch is verified to work properly on all `Beta` machines, the patch can be moved from the `Beta` container up to the `Solaris` container, which causes it to be installed on all machines.

This approach offers several advantages:

- Because `Beta` is contained in `Solaris`, all other fields in the `Solaris` container are acquired by the machines in `Beta`. Therefore, the configurations of these machines will stay as close as

possible to the standard configuration being used by the rest of the environment.

- Because all hosts for testing the new configuration are aggregated into one container, it is easy to keep track of which machines are being used as test cases.

```
Solaris contains {
    north-star;
    dog-star;
    # and many, many others
}
Beta;
}

Beta contains {
    # machines of various
    # hardware configurations
    chorf;
    emerald-city;
}
```

Figure 4: Beta test containment.

```
EthernetInterface Interface {
    acquire mode;    # declared in network
    acquire ethaddr; # declared in hosts
}

SwitchPort switchport {
    SwitchNum switchnum(switch =
        'zaphod.ccs.neu.edu');
    switchnum.is('8/13');
    acquire vlan;
    acquire ether;
    acquire smode;
}

Network CCSNetwork {
    IfMode mode;
    SwitchPortMode smode;
    mode.is('normal');
    smode.is('normal');
}
```

Figure 5: Interface declaration.

- The beta test containment relationship is orthogonal to other containment relationships such as physical location, network (logical) location, and ownership. Therefore, the inclusion of a machine into the beta test container does not affect the usage patterns of the machine.
- When no upgrades are being tested, the machines in the beta container acquire exactly the same settings that machines outside the beta container acquire. Therefore, the addition of the beta container does not affect the environment in any way when it is not being used. In other words, the container becomes completely transparent when nothing is being tested.

While it might seem that the ‘has’ assertion in this example must embody all of the functionality of

Cfengine, it is not nearly that complex. It uses a directory that contains all the Solaris patches currently in use in the environment. In it there is a subdirectory named ‘2.8_recommended.’ Other patches are listed individually. The has assertion simply checks if a patch is installed, and if it isn’t, runs the patch’s install script. The packages field works similarly. This method, of course, cannot handle all of the boundary cases handled by Cfengine.

Reparenting

Modeling context through containment is the central theme of Anomaly. The following example demonstrates how changes in context and containment result in appropriate changes in the network, with a minimum of reconfiguration.

Consider the containment graph in Figure 6. The object declarations for individual hosts and unmanaged containers are not shown, but are similar to the declarations used in Figures 8 and 1. The one unfamiliar object in this graph is Interface, which represents Ethernet interfaces on Unix hosts. Its declaration is shown in Figure 5; it is placed into its containers with copy containment. Figure 5 also shows the declaration of CCSNetwork, which specifies a mode (promiscuous or normal) which is acquired by all interfaces.

Suppose that the administrator of this network wishes to start running host-based IDS software. This can be accomplished without modifying the declarations of individual hosts or interfaces at all. The administrator simply adds an IDS container and reparents the appropriate objects as shown in Figure 7. Now, all IDS-related settings are aggregated into a single container. To make a machine an IDS machine, it needs only to be added to the IDS container. By acquiring settings from the IDS container, the machine receives the IDS packages, the machine’s interface becomes promiscuous, and the switch port attached to the interface is put into spanning mode.

By modeling context through containment, we are able to aggregate control of three network components (host software, host interface, and switch port) into a single point of control. Examples such as this one make changes to configurations atomic (i.e., the desired change can be effected by moving one machine into one container), and thus more maintainable.

The Language

Anomaly is a simple object-oriented declarative language. Its basic components are:

- **Objects.** Objects in Anomaly are used to model components of a network. They can be divided into two categories:
 - *Managed Objects.* Managed objects represent elements of the network that are directly managed by system administrators, e.g., computers, switches, and printers.
 - *Unmanaged Objects.* Unmanaged objects are components of the network that are not

directly managed as part of the network, e.g., buildings, rooms, and research groups.

A simple Anomaly object appears in Figure 8.

```
# unix platform
Unix contains {
    Linux;
    Solaris;
    OpenBSD;
}

Linux contains {
    darkside;
}

OpenBSD contains {
    runningboar;
}

Solaris contains {
    chorff;
    ambler;
    north-star;
}

# a building
Cullinane contains {
    UnixLab;
    MrRoom; # machine room
}

CCSNetwork contains {
    Subnet115;
    Subnet116;
    Subnet118;
}

Subnet115 contains {
    runningboar; # a host
}

Subnet116 contains {
    chorff;
    ambler;
    north-star;
}

Subnet118 contains {
    darkside;
}

ContainmentTemplate (
    QUERY = "SELECT hostname, \
              switchport FROM hosts;"
    NAME = hostname;
) contains {
    copy Interface contains {
        QUERY.switchport;
    }
}
```

Figure 6: Containment relationships in a small network.

- **Fields.** Fields store the configuration data for objects. In Figure 8, fields include hostname, ip, and accesspolicy.
- **Parameters.** Fields may be parameterized. In Figure 8, the field accesspolicy specifies that

/etc/passwd is the location of the file it generates. The purpose of parameters is to tell Anomaly *where* data must go. Another example of a parameter is the name of a network interface object, e.g., hme0 or eth1.

- **Assertions.** In Figure 8, ip = '129.10.117.177'; is an assertion about the value of the field ip. It says: “The value of ip is 129.10.117.177.” Assertions are implemented by the fields that use them, not by the core of Anomaly. Therefore, two different types of fields (e.g., AccessPolicy, Packages) may have completely different implementations of the ‘has’ assertion.

```
Platform IDS {
    IfMode mode;
    SwitchPortMode smode;
    packages.has('snort');
    packages.has('acid');
    mode.is('promiscuous');
    smode.is('spanning');
    acquire packages;
}

CCSNetwork contains {
    IDS;
}

OpenBSD contains {
    IDS;
}

IDS contains {
    runnnigbear;
}
```

Figure 7: IDS declarations.

```
SolarisHost chorff {
    Fqdn hostname;
    IPAddr ip;
    Access accesspolicy('/etc/passwd')
    ip = '129.10.117.177';
    acquire resolv;
    acquire accesspolicy;
    acquire mounts;
}
```

Figure 8: Example object.

- **Acquisitions.** In Anomaly, the keyword acquire signifies the acquisition of a field. Acquisitions tell Anomaly to search for the value of that field in the containers of an object. It is important to note that accesspolicy is an acquired field, yet it has a parameterized declaration. In this example, accesspolicy acquires its value from its containers, but retains the parameters specified by its object. The other acquired fields in this example result in the acquisition of the value *and* the parameters from the containers, since no declaration is specified in the object.
- **Containment declarations.** A simple containment declaration appears in Figure 9. The three

blocks in the example declare that *chorf* and *ambler* are *part of* Subnet116, *chorf* is *in* Room201, and *ambler* is *in* Room129.

- **Database templates.** While language-based configuration approaches have many merits, storing configurations for hundreds of machines in source code form can be unwieldy. Database templates are provided in order to integrate Anomaly with configuration database back-ends, by allowing for the creation of arbitrarily many objects in only a few lines of code.

```
Subnet116 contains {
    chorf;
    ambler;
}

Room201 contains {
    chorf;
}

Room129 contains {
    ambler;
}
```

Figure 9: Example containers.

Semantics of Containment

Objects in Anomaly may have multiple, non-nested containers. In fact, most do. For example, a host usually belongs (at minimum) to both a room and a subnet at the same time. Yet, the subnet is not in the room, and the room is not in the subnet.

During the design process, we considered requiring that the containment graph take the form of a forest whose trees meet only at the leaves. This decision turned out to be unduly restrictive. Therefore, the only requirement Anomaly places on the containment graph is that it be directed and acyclic, that is, objects may not contain themselves, directly or indirectly. Users of Anomaly can construct complex containment graphs using this rule.

When attempting to model certain containment relationships in this way, there are situations in which one would like to use the same object in many locations. For example, if network interfaces are represented by objects, they will most likely be identical across a wide group of machines. However, we cannot simply declare one interface object, and place it in each host. Doing so would create a containment graph like the one depicted in the left half of Figure 10, where the objects in the top layer all contain the same interface object. When the interface object attempts to acquire its netmask, it finds a netmask field in each of its parents, and compilation fails.

To remedy this situation, Anomaly offers *copy containment*. Copy containment reduces the number of duplicate object declarations by allowing a single declaration to be used in any number of places. When a containment relationship is declared as copy containment, the contained object is cloned, and the copy is

placed into the container. This results in the containment graph in the right half of Figure 10, where the gray objects are all clones of an original object. This approach was inspired by *mixins*, an alternative approach to multiple inheritance [9].

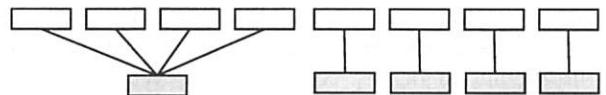


Figure 10: A simple containment graph with and without copy containment.

Semantics of Acquisition

Anomaly uses references to represent containment links between containee and container. When an object explicitly acquires a field, using the *acquire* statement, Anomaly searches for the field in the current object. If the field is not found, the containers of the object are searched recursively, until the field is found.

Because objects may have multiple containers, Anomaly must account for acquisition from multiple containers. If the acquisition search reveals that a variable can acquire its value from two different containers, Anomaly reports an error. This condition is called an acquisition conflict, because two equally valid contexts have been found.

As Anomaly attempts to resolve an acquisition, it may find that an object's container has attempted to acquire the same field. When this scenario occurs, the acquisition in the container is resolved first, and then the acquisition in the containee is resolved, using the value that has now been inserted into the container.

While this intermediate step looks superfluous at first glance, it is necessary to prevent ambiguities about the origin of a field's parameters. Recall that a field may specify its parameters, but acquire its value from the container. Consider three objects that are contained one within the other, like Russian dolls. Suppose that the outer object declares a field called *accesspolicy*, and that the inner two objects acquire that field. In the case where the outer object and the middle object specify different parameters (e.g., */etc/passwd* and */etc/shadow*), the semantics described in the above paragraph force the innermost object to acquire its parameters from the middle object rather than the outer object. In other words, this rule resolves any ambiguity about where a field's parameters are acquired from.

The Dependency Graph

In large configurations, it is both time consuming and inconvenient to re-deploy every configuration on the network when only a few objects have been modified. Therefore, Anomaly uses a simple strategy to deploy configurations only to those objects that may have changed since the last update. This strategy exploits the structure of the containment graph.

The containment graph, in addition to modeling contextual relationships, also models dependency relationships. Using acquisition, a change to the configuration of one object can affect only that object, and any objects contained within. It is not possible for changes in an object to have any effect on its containers. To exploit this invariant, Anomaly keeps track of which objects have been altered since the last successful attempt to deploy configurations. When another attempt is made to deploy the configurations, Anomaly discards all objects that have not been modified, and are not contained within objects that have been modified.⁴

Figure 11 illustrates a modification to the access policy of a computing lab. The left half shows the network before the change is made; the right half shows the network after the change is made. Only objects in the shaded area require re-deployments of their configurations. The shaded area is computed by a simple mark and sweep algorithm.

Checking Types

Anomaly enforces some type constraints during compilation. In particular, it checks assertions for type correctness in the obvious manner. For example, if we declare that some variable represents an IP address and make an assertion about the variable, then Anomaly ensures that the assertion associates a well-formed IP number with the variable. Anomaly does not, however, attempt to enforce constraints at runtime. That is, for all those actions for which it cannot check type constraints during compilation, the configuration transport mechanisms must enforce the coherence of the data access operations (e.g., file access, SNMP set commands). In practice, this means that if a type constraint is violated while Anomaly is generating configurations (i.e., after the source code has been

⁴It is also possible to construct an acquisition graph from the containment graph, in which each edge represents the acquisition of some value. Using this graph, the minimal set of modified machines can be computed. Currently, Anomaly does not compute this minimal modified set.

processed), the configuration will either fail or produce incorrect results. Making Anomaly truly type safe – indeed, exploring what type safety precisely means in this context – is future research.

Templates

Templates allow Anomaly to instantiate many objects at once, using data from a configuration database. Figure 12 shows an example template.

```
Template (
    QUERY = "SELECT hostname FROM hosts \
              WHERE os == 'solaris'";
    NAME = hostname;
) {
    Fqdn hostname;
    hostname = QUERY.hostname;
    acquire access;
}
```

Figure 12: Example template.

Templates consist of two parts. The first part is the query declaration, which appears between the parenthesis at the top of the template declaration. It specifies a query to be issued to the database, and specifies what the identifier (NAME) of each new object will be. In this example, one object is instantiated for each row returned by the specified SQL query, and the object is given the name contained in the ‘hostname’ column of that row.

The second part of the template declaration is the object declaration. It follows all the same rules as a regular object declaration, except that any column of the query result can be referenced by the keyword ‘QUERY,’ as seen on line 8 of Figure 12.

There is also a second type of template, called a containment template, which is used to declare containment relationships. An example of this appears at the end of Figure 12. It follows the same rules as a normal containment declaration, except that it can reference columns in the query result, just like the template above.

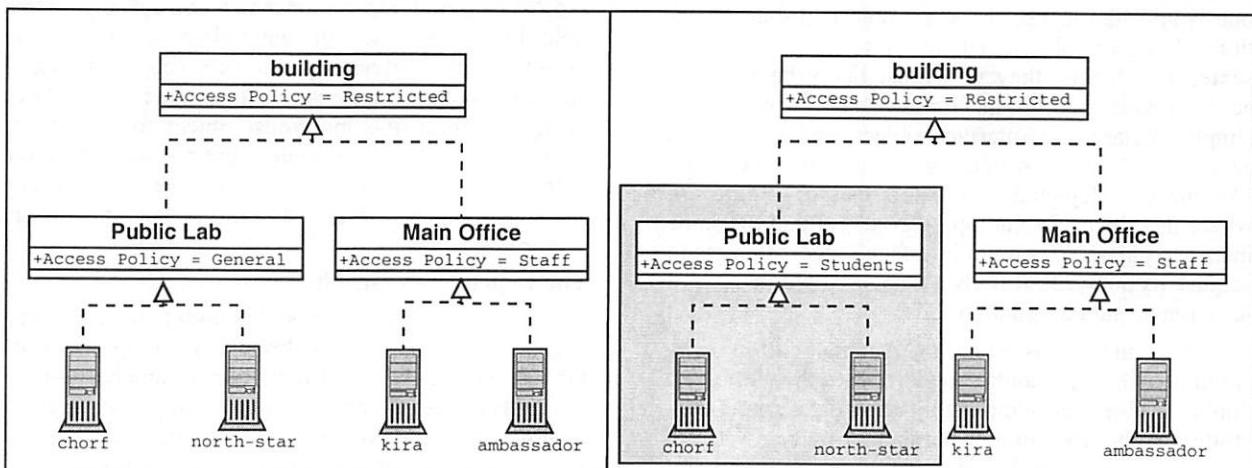


Figure 11: Calculating dependencies after changing access policy for a lab.

Practical System Administration with Anomaly

To see how Anomaly can fit into normal administrative practices, we list several sub-categories of system management, describe them briefly, and show how Anomaly can fit into each area.

Host management is the area of system management embodied by Cfengine. Most of the examples in this paper have focused on this area. Host management tasks lend themselves well to a context-based approach, so Anomaly is best suited to this area of system management. For Anomaly to succeed, however, it must be able to conform itself to existing environments, rather than expecting environments to be built around its notions of containment and context. Fortunately, typical computing environments are ripe with contextual relationships that can be exploited by Anomaly. Therefore, we believe that Anomaly has the potential to be an appropriate addition to existing environments, rather than a foundation for environments that are being rebuilt from the ground up.

Service management includes tasks such as the generation of DNS zone files and DHCP configurations, to name a few. Database driven approaches lend themselves well to these tasks, because the configuration files being produced are often nothing more than a listing of the contents of the database in an obscure format. Anomaly is not particularly well suited to this area of system management, because it often requires data about every object in the environment to be brought together in a single location. A simple query to a configuration database is the appropriate tool for this job. To do the same with an acquisition-based tool would be awkward and inefficient.

User management falls partially under the previous category, because it may involve services such as NIS, LDAP, and Kerberos, but deserves its own category because it also involves resource allocation, specifically home directories and mail spools. User accounts are laden with contextual information, such as the account owner's position within the organization. Anomaly could be used with an existing user management system as a means to model these contextual relationships. Doing so could simplify tasks such as disk quota assignment and account expiration.

Software (or package) management is the area embodied by systems such as Depot, Stow, and RPM. If software packages are installed locally on individual hosts, this category is partially subsumed by host management. However, software is often installed on globally accessible filesystems (e.g., NFS), so software management must be considered separately.

Anomaly is not intended to be a software management system by itself, but it can interface with existing software management tools. For example, an RPM extension to Anomaly would need only to wrap logical assertions around RPM's query based interface. The `has` operator, in this case, would issue a

query to see if a given package was installed on a target machine, and install it if necessary. Here we see that Anomaly can coexist with other systems. There would be no reason to stop using the existing system and rely only on Anomaly. Anomaly would simply be used to model and enforce the contextual relationships affecting software installation (e.g., if a machine is one of the mail servers for an environment, it must have an MTA installed).

Another practical consideration is that of Anomaly's configuration transport mechanism. In practice, Anomaly uses three types of configuration mechanisms: configuration files (e.g., `/etc/passwd` and `/etc/resolv.conf`), configuration scripts (e.g., a script that executes `link` statements), and SNMP set commands. The configuration files are transported to the appropriate machines using ssh and a small helper script that writes the file contents to the appropriate locations. Configuration scripts are copied to a temporary directory and executed on the target machine. SNMP set commands are executed as Anomaly interprets the code. In this sense, Anomaly is primarily a "push" tool, that is, configurations represented in Anomaly are generated on a single host, and then distributed to the managed objects.

A Caveat

Acquisition is not a panacea for all system administration tasks. In particular, an acquisition-based approach does not provide any assistance with unique information in a network. For example, when a host is moved from one subnet to another, its IP address must change,⁵ in addition to its netmask and default route. A natural suggestion is that hosts acquire their IP addresses from their subnet containers, which are responsible for ensuring the uniqueness of each address.

Anomaly's implementation of acquisition cannot facilitate this design, and acquisition in general does not lend itself to this approach. This is true for a number of reasons, the most important one being that to implement such a system, containers would have to retain state about the values that had been handed out to sub-objects. Storing this state would make it impossible to cull unmodified objects by constructing a dependency graph.

We do not believe, however, that the inability to manage unique information constitutes a genuine weakness. The goal of Anomaly is to make it easier to manage information that is common to a group of objects. System administrators are already good at managing unique information, but they do need support to keep common information consistent.

Some First Experiences

We have used Anomaly to manage a lab of Unix workstations, and the switch ports to which they are

⁵Changing IP addresses via a remote configuration management system is problematic for other reasons. Nonetheless, this example illustrates a range of problems in which unique information must be managed.

connected. The workstations run Solaris 2.8, and the switch is a Cisco Catalyst 5500. Using this experimental setup, we were able to construct real containment graphs that effectively modeled our test network. By moving machines to different positions in the containment graph, i.e., by changing their context, we were able to verify how quickly an Anomaly-administered network can be reconfigured when the components of the network change.

The experiments also revealed some weaknesses in Anomaly's configuration transport mechanisms. First, because Anomaly is intended to work with a wide variety of hardware (not necessarily UNIX machines), restricting all objects to an rsh transport mechanism is not feasible. For Anomaly to be extended to other operating systems, we need to design a universal system for configuration transport. Second, to overcome the difficulties of performing changes to networking configurations (e.g., IP addresses, netmasks), Anomaly should produce floppy disks (or other removable media) to transport basic configuration data. We hope to address these problems with future research.

Project Status and Source Code Availability

As of this writing, Anomaly is not ready for widespread use. Its most pressing need is for the development of a large and portable suite of administrative modules. Currently, Anomaly has modules for managing Solaris hosts in an NIS/NFS environment, and a module for managing certain aspects of Cisco Catalyst series switches (specifically, VLAN membership and port security).

Some of the source code listings in this paper show components of Anomaly that are not stable as of this writing for illustrative purposes. Specifically, database templates, the 'packages' field, and the 'patches' field are only partially implemented.

The latest source code, along with current information about Anomaly, is available at <http://www.ccs.neu.edu/home/mlogan/anomaly/>.

Author Information

Mark Logan is a recent graduate (BS) of Northeastern University's College of Computer Science. The work presented in this paper is his senior thesis. While studying for his degree, he spent approximately three years working for CCS in various capacities, primarily as a system administrator. He can be reached electronically at mlogan@ccs.neu.edu.

Matthias Felleisen received his PhD in 1987 from D. P. Friedman (Indiana University), spent 14 years at Rice University as a professor, and joined Northeastern University as a Trustee Professor of Computer Science last year. He is interested in all aspects of program design and programming languages.

David N. Blank-Edelman is the Director of Technology at the Northeastern University College of

Computer Science and the author of Perl for System Administration (O'Reilly). He has spent the last 15 years as a system/network administrator in large multi-platform environments and has served as Senior Technical Editor for the Perl Journal. He has also written many magazine articles on world music.

References

- [1] Evard, R., "An Analysis of UNIX System Configuration," *Proceedings of the Eleventh Systems Administration Conference (LISA XI)*, USENIX Association, Berkeley, CA, p. 179, 1997.
- [2] Anderson, P., "Towards a High-Level Machine Configuration System," *Proceedings of the Eighth Systems Administration Conference (LISA VIII)*, USENIX Association, Berkeley CA, p. 19, 1994.
- [3] Finke, J., "An Improved Approach for Generating Configuration Files from a Database," *Proceedings of the 14th Systems Administration Conference (LISA XIII)*, USENIX Association, Berkeley CA, p. 29, 2000.
- [4] Burgess, M., "A Site Configuration Engine," *Computing Systems*, Vol. 8, No. 1, MIT Press, Cambridge MA, p. 309, Winter 1995.
- [5] Couch, A. and M. Gilfix, "It's Elementary, Dear Watson: Applying Logic Programming to Convergent Systems Management Processes," *Proceedings of the 15th Systems Administration Conference (LISA XV)*, USENIX Association, Berkeley CA, p. 123, 2001.
- [6] Holgate, M. and W. Partain, "The Arusha Project: A Framework for Collaborative Unix System Administration," *Proceedings of the 15th Systems Administration Conference (LISA XV)*, USENIX Association, Berkeley CA, pp. 187-198, 2001,
- [7] Hagemark, B. and K. Zadeck, "Site: A Language and System for Configuring Many Computers as One Computer Site," *Proceedings of the Workshop on Large Installation Systems Administration III*, USENIX Association, Berkeley CA, p. 1, 1989.
- [8] Gil, J. and D. Lorenz, "Environmental Abstraction – A New Inheritance-Like Abstraction Mechanism," *Proceedings of the Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 214-231, October 1996.
- [9] Flatt, M., S. Krishnamurthi, and M. Felleisen, "Classes and Mixins," *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 171-183, January 1998.
- [10] Ungar, D. and R. Smith, "Self: The Power of Simplicity," *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, pp. 227-242, December 1987.

A Simple Way to Estimate the Cost of Downtime

David A. Patterson – University of California at Berkeley

ABSTRACT

Systems that are more dependable and less expensive to maintain may be more expensive to purchase. If ordinary customers cannot calculate the costs of downtime, such systems may not succeed because it will be difficult to justify a higher price. Hence, we propose an easy-to-calculate estimate of downtime.

As one reviewer commented, the cost estimate we propose “is simply a symbolic translation of the most obvious, common sense approach to the problem.” We take this remark as a complement, noting that prior work has ignored pieces of this obvious formula.

We introduce this formula, argue why it will be important to have a formula that can easily be calculated, suggest why it will be hard to get a more accurate estimate, and give some examples.

Widespread use of this obvious formula can lay a foundation for systems that reduce downtime.

Introduction

It is time for the systems community of researchers and developers to broaden the agenda beyond performance. The 10,000X increase in performance over the last 20 years means that other aspects of computing have risen in relative importance. The systems we have created are fast and cheap, but undependable. Since a portion of system administration is dealing with failures [Anderson 1999], downtime surely adds to the high cost of ownership.

To understand why they are undependable, we conducted two surveys on the causes of downtime. In our first survey, Figure 1 shows data we collected failure data on the U. S. Public Switched Telephone Network (PSTN) [Enriquez 2002]. It shows the percentage of failures due to operators, hardware failures, software failures, and overload for over 200 outages in 2000. Although not directly relevant to computing systems, this data set is very thorough in the description of the problem and the impact of the outages. In our second study, Figure 2 shows data we collected failure from three Internet sites [Oppenheimer 2002]. The surveys are notably consistent in their suggestion that operators are the leading cause of failure.

Collections of failure data often ignore operator error, as it often requires asking operators if they think they made an error. Studies that are careful about how they collect data do find results that are consistent with these graphs [Gray 1985, Gray 1990, Kuhn 1997, Murphy 1990, Murphy 1995].

Improving dependability and lowering cost of ownership are likely to require more resources. For example, an undo system for operator actions would need more disk space than conventional systems. The marketplace may not accept such innovations if

products that use them are more expensive and the subsequent benefits cannot be quantified by lower cost of ownership. Indeed, a common lament of computer companies that customers may complain about dependability, but are unwilling to pay the higher price of more dependable systems. The difficulty of measuring cost of downtime may be the reason for apparently irrational behavior.

Hence, this paper, which seeks to define a simple and useful estimate of the cost of unavailability.

Estimating Revenue and Productivity

Prior work on estimating the cost of downtime is usually measuring the loss of revenue for online companies or other services that cannot possibly function if their computers are down [Kembel 2000]. Table 1 is a typical example.

Brokerage operations	\$6,450,000
Credit card authorization	\$2,600,000
Ebay	\$225,000
Amazon.com	\$180,000
Package shipping services	\$150,000
Home shopping channel	\$113,000
Catalog sales center	\$90,000
Airline reservation center	\$89,000
Cellular service activation	\$41,000
On-line network fees	\$25,000
ATM service fees	\$14,000

Table 1: Cost of one hour of downtime. From *InternetWeek* 4/3/2000 and based on a survey done by Contingency Planning Research. [Kembel 2000].

Such companies are not the only ones that lose revenue if there is an outage. More importantly, such a table ignores the loss to a company of wasting the time of employees who cannot get their work done during an outage, even if it does not affect revenue. Thus, we need a formula that is easy to calculate so that administrators and CIOs in any institution can determine the costs of outage. It should capture both the cost of lost productivity of employees and the cost of lost revenue from missed sales.

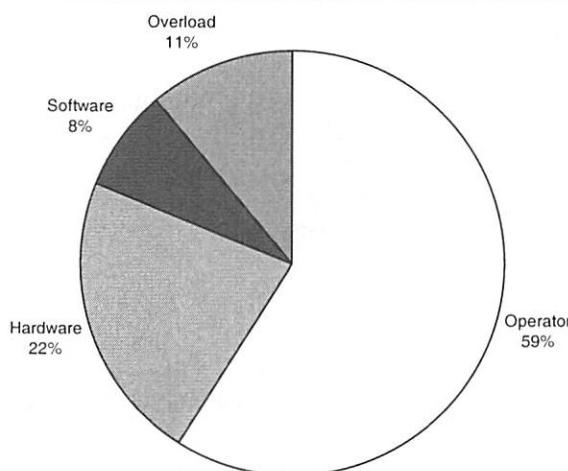


Figure 1: Percentage of failures by operator, hardware, software, and overload for PSTN. The PSTN data measured blocked calls during an outage in the year 2000. (This figure does not show vandalism, which is responsible for 0.5% of blocked calls.) We collected this data from the FCC; it represents over 200 telephone outages in the U. S. that affected at least 30,000 customers or lasted at least 30 minutes. Rather than only reporting outages, telephone switches record the number of attempted calls blocked during an outage, which is an attractive metric. The figure does not include environmental causes, which are responsible for 1% of the outages.

We start with the formula, and then explain how we derived it:

$$\begin{aligned} \text{Estimated average cost of 1 hour of downtime} = \\ \text{Empl. costs/hour} * \% \text{ Empl's affected by outage} + \\ \text{Avg. Rev./hour} * \% \text{ Rev. affected by outage} \end{aligned}$$

Employee costs per hour is simply the total salaries and benefits of all employees per week divided by the average number of working hours per month. *Average revenue per hour* is just the total revenue of an institution per month divided by average number of hours per week an institution is open for business. Note that this term includes two factors: revenue associated with a web site and revenue supported by the internal information technology infrastructure. We believe these employee costs and revenue are not too difficult to calculate, especially since they are input to an estimate, and hence do not have to be precise to the last penny.

For example, publicly traded companies must report their revenue and expenses every quarter, so quarterly statements have some of the data to calculate these terms. Although revenue is easy to find, employee costs are typically not reported separately. Fortunately, they report the number of employees, and you can estimate the cost per employee. The finance department of smaller companies must know both these terms to pay the bills and issue paychecks. Even departments in public universities and government agencies without conventional revenue sources have their salaries in the public record.

The other two terms of the formula – fraction employees and fraction revenue affected by outage – just need to be educated guesses or ranges that make sense for your institution.

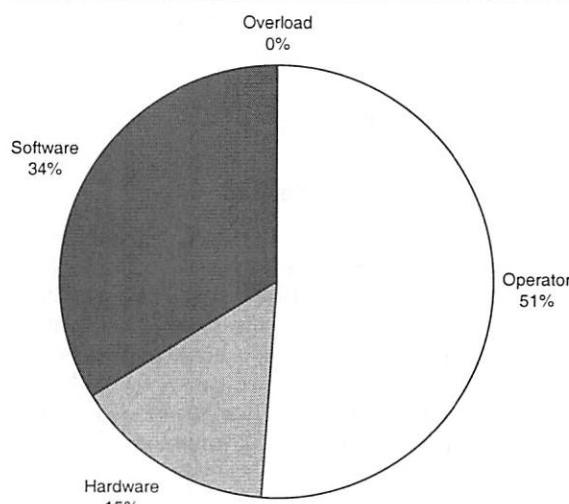


Figure 2: Percentage of failures by operator, hardware, software, and overload for three Internet sites. Note that the mature software of the PSTN is much less of a problem than Internet site software, yet the Internet sites have such frequent fluctuations in demand that they have overprovisioned sufficiently so that overload failures are rare. The Internet site data measured outages in 2001. We collected this data from companies in return for anonymity; it represents six weeks to six months of services with 500 to 5000 computers. Also, 25% of outages had no identifiable cause, and are not included in the data. One reviewer suggested the explanation was “nobody confessed;” that is a plausible interpretation.

Is Precision Possible?

As two of the four terms are guesses, the estimate clearly is not precise and open to debate. Although we all normally strive for precision, it may not be possible here. To establish the argument for systems that may be more expensive to buy but less expensive to own, administrators only need to give an example of what the costs might be using the formula

above. CIOs could decide on their own fractions in making their decisions.

The second point is that much more effort may not ultimately lead to a precise answer, for there are some questions that will be very hard to answer. For example, depending on the company, one hour of downtime may not lead to lost revenue, as customers may just wait and order later. In addition, employees may simply do other work for an hour that does not involve a computer. Depending on centralization of services, an outage may only affect a portion of the employees. There may also be different systems for employees and for sales, so an outage might affect just one part or the whole company. Finally, there is surely variation in cost depending when an outage occurs; for most companies, outages Sunday morning at 2 AM probably has little impact on revenue or employee productivity.

Before giving examples, we need to qualify this estimate. It ignores the cost of repair, such as the cost of overtime by operators or bringing in consultants. We assume these costs are small relative to the other terms. Second, the estimate ignores daily and seasonal variations in revenue, as some hours are more expensive than others. For example, a company running a lottery is likely to have rapid increase in revenue as the deadline approaches. Perhaps a best case and worst-case cost of downtime might be useful as well as average, and its clear how to calculate them from the formula.

Example 1: University of California at Berkeley EECS Department

The Electrical Engineering and Computer Science (EECS) department at U. C. Berkeley does not sell products, and so there is no revenue to lose in an outage. As we have many employees, the loss in productivity could be expensive.

The employee costs have two components: those paid for by state funds and those paid for by external research funds. The state pays 68 full-time staff collective salaries and benefits of \$403,130 per month. This is an annual salary and benefits of \$71,320. These figures do not include the 80 faculty, who are paid approximately \$754,700 per month year round, including benefits. During the school year external research pays 670 full-time and part-time employees \$1,982,500, including benefits. During the summer, both faculty and students can earn extra income and some people are not around, so the numbers change to 635 people earning \$2,915,950. Thus, the total monthly salaries are \$3,140,330 during the year and \$4,073,780 during the summer.

If we assume people worked 10 hours a working day, the Employee costs per hour is \$14,780 during the school year and \$19,170 during the summer. If we assumed people worked 24 hours a day, seven days a week, the costs would change to \$4300 and \$5580.

If EECS file servers and mail servers have 99% availability, that would mean seven hours of downtime per month. If half of the outages affected half the employees, the annual cost in lost productivity would be \$250,000 to \$300,000.

Since I was a member of EECS, it only took two emails to collect the data.

Example 2: Amazon

Amazon has revenue as well as employee costs, and virtually all of their revenue comes over the net. Last year their revenue was \$3.1B and it has 7744 employees for the last year. This data is available at many places; I got it from Charles Schwab. Alas, annual reports do not break down the cost of employees, just the revenue per employee. That was \$400,310. Let's assume that Amazon employee salaries and benefits are about 20% higher than University employees at, say, \$85,000 per year. Then Employee costs per hour working 10 hours per day, five days per week would be \$258,100 and \$75,100 for 24x7. Revenue per hour is \$353,900 for 24x7, which seems the right measure for an Internet company. Thus, an outage during the workweek that affected 90% employees and 90% of revenue streams could cost Amazon about \$550,000 per hour.

We note that employee costs are a significant fraction of revenue, even for an Internet company like Amazon. One reason is the Internet allows revenue to arrive 24x7 while employees work closer to more traditional workweeks.

Example 3: Sun Microsystems

Sun Microsystems probably gets little of its income directly over the Internet, since it has an extensive sales force. That sales force, however, relies extensively on email to record interactions with customers. The collapse of the World Trade Center destroyed several mail servers in the New York area and many sales records were lost. Although its not likely that much revenue would be lost if the Sun site went down, it could certainly affect productivity if email were unavailable, as the company's nervous systems appears to be email.

In the last year, Sun's revenue was \$12.5B and it had 43,314 employees. Let's assume that the average salary and benefits are \$100,000 per employee, as Sun has a much large fraction of its workforce in engineering than does Amazon. Then employee costs per hour working 10 hours per day, five days per week would be \$1,698,600 and \$494,500 for 24x7. Since Sun is a global company, perhaps 24x5 is the right model. That would make the costs \$694,100 per hour. Revenue per hour is \$1,426,900 for 24x7 and \$2,003,200 for 24x5, with the latter the likely best choice.

Let's assume that a workweek outage affected 10% of revenue that hour and 90% of the employees. The cost per hour would about \$825,000, with three-fourths of the cost being employees.

Conclusion

The goal of this paper is to provide an easy to calculate estimate of the average cost of downtime so as to justify systems that may be slightly more expensive to purchase but potentially spend much less time unavailable. It is important for administrators and CIOs to have an easy to use estimate to set the range of costs of outages to make them more likely to take it into consideration when setting policies and acquiring systems. If it were hard to calculate, few people would do it.

Although a simple estimate, we argue that a much more time-consuming calculation may not shed much more insight, as it is very hard to know how many consumers will simply reorder when the system comes back versus go to a competitor, or how many employees will do something else productive while the computer is down.

We see that employee costs, traditionally ignored in such estimates, are significant even for Internet companies like Amazon, and dominate the costs of more traditional organizations like Sun Microsystems. Outages at universities and government organizations can still be expensive, even without a loss of a significant computer-related revenue stream.

In addition to this estimate, there are may be indirect costs to outages that can be as important to the company as these more immediate costs. Outages can lead to management overhead as the IT department is blamed for every possible problem and delay throughout the company. Company morale can suffer, reducing everyone's productivity for periods that far exceed the outage time. Frequent outages can lead to a loss of confidence in the IT team and its skills. Such change in stature could eventually lead to individual departments hiring their own IT people, which lead to direct costs.

As many researchers are working on these solutions to the dependability problems [IBM 2000, Patterson 2002], our hope is that these simple estimates can help organizations justify systems that are more dependable, even if a bit more expensive.

References

- [Anderson 1999] Anderson, E. and D. Patterson, "A Retrospective on Twelve Years of LISA Proceedings," *Proc. 13th Systems Administration Conference – LISA 1999*, Seattle, Washington, p. 95-107, Nov 1999,
- [Enriquez 2002] Enriquez, P., A. Brown, and D. Patterson, "Lessons from the PSTN for Dependable Computing," *Workshop on Self-Healing, Adaptive and self-MANaged Systems (SHAMAN)*, New York, NY, June 2002.
- [Gray 1985] Gray, J., *Why Do Computers Stop and What Can Be Done About It?* TR-85.7, Tandem Computers, 1985.
- [Gray 1990] Gray, J., *A Census of Tandem System Availability Between 1985 and 1990*. TR-90.1, Tandem Computers, 1990.
- [IBM 2000] *Autonomic Computing*, <http://www.research.ibm.com/autonomic/>, 2000.
- [Kembel 2000] Kembel, R., *Fibre Channel: A Comprehensive Introduction*, 2000.
- [Kuhn 1997] Richard Kuhn, D., "Sources of Failure in the Public Switched Telephone Network," *IEEE Computer*, Vol. 30, No. 4, <http://hissa.nist.gov/kuhn/pstn.html>, April, 1997.
- [Murphy 1995] Murphy, B. and T. Gant, "Measuring System and Software Reliability Using an Automated Data Collection Process," *Quality and Reliability Engineering International*, Vol. 11, pp. 341-353, 1995.
- [Murphy 2000] Murphy, B., "Windows 2000 Dependability," *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, June 2000.
- [Oppenheimer 2002] Oppenheimer, D. and D. A. Patterson. "Studying and using failure data from large-scale Internet services," *Tenth ACM SIGOPS European Workshop*, Saint-Emilion, France, September 2002.
- [Patterson 2002] Patterson, D., A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft, *Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies*, U. C. Berkeley Computer Science Technical Report UCB//CSD-02-1175, March 15, 2002.

Defining and Monitoring Service Level Agreements for Dynamic e-Business

Alexander Keller and Heiko Ludwig – IBM T. J. Watson Research Center

ABSTRACT

Fueled by the growing acceptance of the Web Services Architecture, an emerging trend in application service delivery is to move away from tightly coupled systems towards structures of loosely coupled, dynamically bound systems to support both long and short business relationships. It appears highly likely that the next generation of e-Business systems will consist of an interconnection of services, each provided by a possibly different service provider, that are put together on an “on demand” basis to offer an end to end service to a customer.

Such an environment, which we call Dynamic e-Business (DeB), will be administered and managed according to dynamically negotiated Service Level Agreements (SLA) between service providers and customers. Consequently, system administration will increasingly become SLA-driven and needs to address challenges such as dynamically determining whether enough spare capacity is available to accommodate additional SLAs, the negotiation of SLA terms and conditions, the continuous monitoring of a multitude of agreed-upon SLA parameters and the troubleshooting of systems, based on their importance for achieving business objectives.

A key prerequisite for meeting these goals is to understand the relationship between the cost of the systems an administrator is responsible for and the revenue they are able to generate, i.e., a model needs to be in place to express system resources in financial terms. Today, this is usually not the case.

In order to address some of these problems, this paper presents the *Web Service Level Agreement (WSLA)* framework for defining and monitoring SLAs in inter-domain environments. The framework consists of a flexible and extensible language based on the XML schema and a runtime architecture based on several SLA monitoring services, which may be outsourced to third parties to ensure a maximum of accuracy.

WSLA enables service customers and providers to unambiguously define a wide variety of SLAs, specify the SLA parameters and the way how they are measured, and tie them to managed resource instrumentations. A Java-based implementation of this framework, termed *SLA Compliance Monitor*, is publicly available as part of the IBM Web Services Toolkit.

Introduction and Motivation

The pervasiveness of the Internet provides a platform for businesses to offer and buy electronic services, such as financial information, hosted services, or even applications, that can be integrated in a customer’s application architecture. Upcoming standards for the description and advertisement of, as well as the interaction with, online services promise that organizations can integrate their systems in a seamless manner. The Web Services framework [16] provides such an integration platform, based on the WSDL service interface description language, the UDDI directory service [31] and, for example, SOAP over HTTP as a communication mechanism. Web Services provide the opportunity to dynamically bind to services at runtime, i.e., to enter (and dismiss) a business relationship with a service provider on a case-by-case basis, thus creating an infrastructure for *dynamic e-Business* [14].

Dynamic e-Business implies dynamics several orders of magnitude higher than found in traditional

corporate networks. Moreover, a service relationship also constitutes a business relationship between independent organizations, defined in a contract.

An important aspect of a contract for IT services is the set of Quality of Service (QoS) guarantees a service provider gives. This is commonly referred to as a service level agreement (SLA) [32, 17]. Today, SLAs between organizations are used in all areas of IT services – in many cases for hosting and communication services but also for help desks and problem resolution.

Furthermore, the IT parameters for which Service Level Objectives (SLO) are defined come from a variety of disciplines, such as business process management, service and application management, and traditional systems and network management. In addition, different organizations have different definitions for crucial IT parameters such as Availability, Throughput, Downtime, Bandwidth, Response Time, etc. Today’s SLAs are plain natural language documents. Consequently, they must be manually provisioned and monitored, which is very expensive and slow.

The definition, negotiation, deployment, monitoring and enforcement of SLAs must become – in contrast to today's state of the art – an automated process. This poses several challenges for the administration of shared distributed systems, as found in Internet Data Centers, because administrative tasks become increasingly dynamic and SLA-driven.

The objective of this paper is to present the *Web Service Level Agreement (WSLA)* framework as an approach to deal with these problems; it provides a flexible, formal language and a set of elementary services for defining and monitoring SLAs in dynamic e-Business environments.

The paper is structured as follows: In the next section, we describe the underlying principles of our work, analyze the requirements of dynamic e-Business on system administration tasks and on the WSLA framework. We also describe the relationships of our work to the existing state of the art. The WSLA runtime architecture, described later, provides mechanisms for accessing resource metrics of managed systems and for defining, monitoring and evaluating SLA parameters according to an SLA specification. We subsequently introduce the WSLA language by means of several examples. It is based on the XML Schema and allows parties to define QoS guarantees for electronic services and the processes for monitoring them. Finally, the last section concludes the paper and gives an overview of our current work.

Principles of the WSLA Framework

Service level management has been the subject of intense research for several years now and has reached a certain degree of maturity. However, despite initial work in the field (see, e.g., [2]), the problem of establishing a generic framework for service level management in cross-organizational environments remains unsolved yet. In this section, we introduce the terminology and describe the fundamental principles, which will be used throughout this paper. Subsequently, focusing on SLA-driven system administration, we derive the requirements of the WSLA language and its runtime architecture.

Terminology

There are various degrees to which extent a service customer is willing to accept the parameters offered by the service provider. Metric- and SLA-related information appears at various tiers of a distributed system, as depicted in Figure 1.

- **Resource Metrics** are retrieved directly from the managed resources residing in the service provider's tier, such as routers, servers, middleware and instrumented applications. Typical examples are the well-known MIB variables of the IETF Structure of Management Information (SMI) [21], such as counters and gauges.
- **Composite Metrics** are created by aggregating several resource (or other composite) metrics

according to a specific algorithm, such as averaging one or more metrics over a specific amount of time or by breaking them down according to specific criteria (e.g., top 5%, minimum, maximum etc.). This is usually being done within the service providers' domain but can be outsourced to a third-party measurement service as well. Composite metrics are exposed by a service provider by means of a well-defined (usually HTTP or SOAP based) interface for further processing.

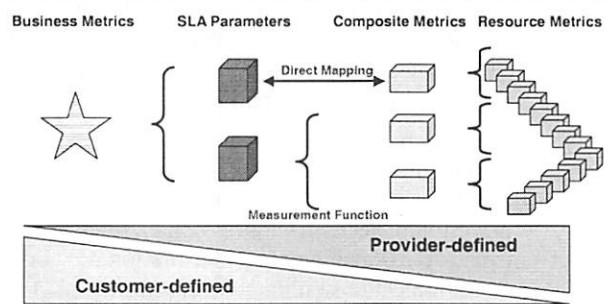


Figure 1: Aggregating business metrics, SLA parameters and metrics across different organizations.

- **SLA Parameters** put the metrics available from a service provider into the context of a specific customer and are therefore the core part of an SLA. In contrast to the previous metrics, every SLA parameter is associated with high/low watermarks, which enables the customer, provider, or a designated third party to evaluate the retrieved metrics whether they meet/exceed/fall below defined service level objectives. Consequently, every SLA parameter and its permitted range are defined in the SLA. It makes sense to delegate the evaluation of SLA parameters against the SLOs to an independent third party; this ensures that the evaluation is objective and accurate.
- **Business Metrics** relate SLA parameters to financial terms specific to a service customer (and thus are usually kept confidential by him). They form the basis of a customer's risk management strategy and exist only within the service customer's domain. It should be noted that a service provider needs to perform a similar mapping to make sure the SLAs he is willing to satisfy are in accordance with his business goals.

The WSLA framework presented in this paper is capable of handling all four different parameter types; apart from the latter, they relate directly to systems management tasks and are our main focus. However, the flexible mechanism for composing SLAs can be easily extended to accommodate business metrics.

Scenarios for SLA Establishment

Often, it is not obvious to draw a line between the aforementioned parameter types, in particular

between Composite Metrics and SLA Parameters. Therefore, we assume that every parameter related to a customer and associated with a guaranteed value range is considered an SLA parameter, which is supposed to be part of an SLA. However, this distinction is also highly dependent on the extent a customer requires the customization of metrics exposed by the service provider (or a third-party measurement service) – and how much he is willing to pay for it. This, in turn, depends on the degree of customization the provider is willing to apply to its metrics. The following scenarios describe various scenarios how SLAs may be defined:

- 1. A customer adopts the data exposed by a service provider without further refinement**
This is often done when the metrics reflect good common practice, cannot be modified by the customer or are of small(er) importance to him. In this case, the selected metrics become the SLA parameters and thus integral parts of the SLA. Examples are: *length of maintenance intervals* or *backup frequency*.
- 2. The customer requests that collected data is put into a meaningful context**
A customer is probably not interested in the overall availability of a provider's data center, but needs to know the availability of the specific cluster within the data center on which his applications and data are hosted. A provider's data collection algorithm therefore needs – at least – to take into account for which customer the data is actually collected. A provider may decide to offer such preprocessed data, such as: *Availability of the server cluster hosting customer X's web application*.
- 3. The customer requests customized data that is collected according to his specific requirements**
While a solution to item 2 can still be reasonably static (changes tend to happen rarely and the nature of the modifiable parameters can be anticipated reasonably well), the degree of choice for the customer can be taken a step further by allowing him to specify arbitrary parameters, e.g., the input parameters of a data collection algorithm.
This implies that a service provider needs to have a mechanism in place that allows a customer to provide these input parameters – preferably at runtime, e.g., *The average load of a server hosting the customer's website should be sampled every 30 seconds and collected over 24 hours*. Note that a change of these parameters may result in a change of the terms and conditions of an SLA, e.g., when a customer chooses sampling intervals that are likely to impact the performance of the monitored system; eventually, this may entail the violation of SLAs the service provider has with other customers.
- 4. The customer specifies how data is collected**
This means that he defines – in addition to the

metrics and input parameters – the data collection algorithm itself. This is obviously the most extreme case and seems fairly unlikely. However, large customers may insist of getting access to very specific data that is not part of the standard set, e.g., a customer may want to know which employees of a service provider had physical access to the systems hosting his data and would like to receive a daily log of the badge reader.

This means that – in addition to the aforementioned extension mechanisms – a service provider needs to have a mechanism in place that allows him to introduce new data collection mechanisms without interrupting his management and production systems.

While the last case poses the highest challenge on the programmability of the monitoring system, a service provider benefits greatly from a management system being capable of handling such flexible SLAs because all the former situations are special cases of the latter. It also addresses the extreme variability of today's SLAs. Sample SLAs we analyzed clearly indicate that there is a need for defining a mechanism that allows to unambiguously specify the data collection algorithm. Also, it should be noted that the different possibilities of specifying service level objectives are not mutually exclusive and may all be specified within the same SLA.

SLA-driven System Administration

Now that we have introduced the concepts of SLA management in a dynamic e-Business environment, we are able to derive its implications on systems administration and management. While it is clear that the very high dynamics of the establishment/dismissal of business relationships and the resulting allocation/deallocation of system resources to different users alone is a challenge on its own, we have found several other issues that are likely to impact how system administration is done in such an environment. The way we see the tasks of a system administrator evolve are described in the following subsections.

Express System Resources in Financial Terms

While system administrators usually have an awareness of the costs of the systems they are administering, the need to assign prices to the various resources on a very fine-grained basis will certainly increase. For quite some time, it has been common practice in well-run multi-customer data centers to account for CPU time, memory usage and disk space usage on a per-user basis. What will become increasingly important in SLA-driven system administration is the monitoring, accounting and billing of aggregated QoS parameters such as response time, throughput and bandwidth, which need to be collected across a variety of different systems that are involved in a multi-tiered server environment.

Having such a fine-grained accounting scheme in place is the prerequisite for defining SLOs, together with associated penalties or bonuses. In addition, the business impact of an outage or delay on the customer needs to be assessed. While the latter is mainly relevant to a service customer, a system administrator on the service provider side will need an even better understanding of the cost/benefit model behind the services offered to a customer. As a sidenote, the ability to offer measurement facilities for fine-grained service parameters is likely to become a distinguishing factor among service providers.

Involvement in SLA Negotiation

The technical expertise of a system administrator is likely to play an increasing role in an area that is currently confined to business managers and lawyers: The negotiation of SLAs terms. While current SLAs are dominated by legal terms and conditions, it will become necessary in an environment where resources are shared among different customers (under a variety of SLAs) to evaluate whether enough spare capacity is available to accommodate an additional SLA that asks for a specific amount of resources without running into the risk that the resources become overallocated if a customer's demand increases. While complex resource allocation schemes will probably not be deployed in the near future, an administrator nevertheless needs to have an understanding of the safety margins he must take into account when accepting new customers.

A related problem is to evaluate whether additional load due to SLA measurements is acceptable or not: While it may well be the case that enough capacity is available to accommodate the workload resulting from the service usage, overly aggressive SLA measurement algorithms may have a detrimental impact on the overall workload a system can handle. An extreme example for this is a customer whose application resides on a shared server and who would like to have the availability of the system being probed every few seconds. In this case, an SLA may either need to be rejected due to the additional workload, or the price for carrying out the measurements will need to be adjusted accordingly.

Classify Customers According to Revenue

The previous discussions make it clear that a service provider's approach to SLA-driven management entails the definition of enterprise policies that classify customers, e.g., according to the profit margins or their degree of contribution to a service provider's overall revenue stream. The involvement of system administrators in the process of policy definition and enforcement is a consequence of having both a high degree of technical understanding and insight into the business: First, this expertise is needed to determine which policies are reasonable and enforceable.

Second, once the policies are defined, it is up to the administrator to enforce them, e.g., if the resource

capacity becomes insufficient because of increased workload of a high-paying customer, lower-paying customers may be starved out if the penalties associated with their SLAs can be offset by the increased gains from providing additional capacity to a higher-paying customer. Third, it should be noted that such a behavior adds an interesting twist to the problem determination schemes an administrator uses: The non-functioning of a customer's system may not necessarily be due to a technical failure, but may well be the consequence of a business decision.

Fix Outages According to Classification

The establishment of policies and the classification of customers also has implications on how system outages are addressed. Traditionally, system administrators are trained to address the most severe outages first. This may change if a customer classification scheme is in place, because then the system whose downtime or decreased level of service is the most expensive for the service provider will need to be fixed first. Outages are likely going to be classified not according to their technical severity, but rather based on their business impact.

Lessons Learned From Real-Life SLAs

A suitable SLA framework for Web Services must not constrain the parties in the way they formulate their clauses but instead allow for a high degree of flexibility. A management tool that implements only a non-modifiable textbook definition of availability would not be considered helpful by today's service providers and their customers.

Our studies of close to three dozen SLAs currently used throughout the industry in the areas of application service provisioning (ASP) [1], web hosting and information technology (IT) outsourcing have revealed that even if seemingly identical SLA parameters are being defined, their semantics vary greatly.

While some service providers confine their definition of "application availability" to the network level of the hosting system ("user(s) being able to establish a TCP connection to the appropriate server"), others refer to the application that implements the service ("Customer's ability to access the software application on the server"). Still others rely on the results obtained from monitoring tools ("the application is accessible if the server is responding to HTTP requests issued by a specific monitoring software"), while another approach uses elaborate formulas consisting of various metrics, which are sampled over fixed time intervals.

These base clauses are then usually annotated with exceptions, such as maintenance intervals, weekend/holiday schedules, or even the business impact of an outage ("An outage has been detected by the ASP but no material, detrimental impact on the customer has occurred as a result"). The latter example, in particular, illustrates the disconnect between the people

involved in the negotiation and establishment of an SLA (usually business managers and lawyers) and the ones who are supposed to enforce it (system administrators). One way of closing this gap is to enable system administrators to become involved in the negotiation of an SLA by providing them with a tool able to create a legal document, namely the SLA.

It is important to keep in mind that, while the nature of the clauses may differ considerably among different SLAs, the general structure of all the different SLAs remains the same: Every analyzed SLA contains

- the involved parties,
- the SLA parameters,
- the metrics used as input to compute the SLA parameters,
- the algorithms for computing the SLA parameters,
- the service level objectives and the appropriate actions to be taken if a violation of these SLOs has been detected.

This implies that there is a way to come up with an SLA language that can be applied to a multitude of bilateral customer/provider relationships.

WSLA Design Goals

In this section, we will derive – based on the above discussions – the requirements the WSLA framework needs to address.

Ability to Accommodate a Wide Variety of SLAs

In the introduction of this paper, we have stressed the point that SLAs, their parameters and the SLOs defined for them are extremely diverse. One approach to deal with this problem (e.g., as it is done today for simple consumer Web hosting services) is to narrow down the “universe of discourse” to a few well-understood terms and to limit the possibilities of choosing arbitrary QoS parameters through the use of SLA templates [24]. SLA templates include several automatically processed fields in an otherwise natural language-written SLA. However, the flexibility of this approach is limited and only suitable for a small set of variants of the same type of service using the same QoS parameters and a service offering that is not likely to undergo changes over time. In situations where service providers must address different SLA requirements of their customers, they need a more flexible formal language to express service level agreements and a runtime architecture comprising a set of services being able to interpret this language.

Leverage Work in the B2B Area for SLA Negotiation and Creation

Architectural components and language elements related to SLA negotiation, creation and deployment should leverage existing concepts developed in the electronic commerce and B2B area. In particular, the applicability of automated negotiation mechanisms, e.g., currently being developed within the scope of the OASIS/ebXML [6] Collaboration Profiles and

Agreements initiative [7], should be applicable to the negotiation of SLAs as well. A vast amount of work on electronic contracts [25, 22], contract languages [12] and contract negotiation has been carried out in the electronic commerce and B2B arena [4]. We later describe our usage of obligations, a concept widely used in e-commerce, for monitoring SLAs.

Apply the “Need to Know” Principle to SLA Deployment

For each service provider and customer relationship, several instances of a service may exist. The functionality of computing SLA parameters or evaluating contract obligations may be split, e.g., among multiple measurement or SLO evaluation services, each provided by a different organization. It is therefore important that every service instance receives only the part of the contract it needs to know to carry out its task. Since it may be possible that a contractual party delegates the same task (such as measurements) to several different third party services (in order to be able to cross-check their results), different service instances may not be aware of other instances. This implies that every party involved in the SLA monitoring process receives only the part of the SLA that is relevant for him. We present our approach for dealing with this problem later.

Another major issue that underlines the importance of this “Need to know” principle are the privacy concerns of the various parties involved in an inter-domain management scenario: A service provider is, in general, neither interested in disclosing which of his business processes have been outsourced to other providers, nor the names of these providers. On the other hand, customers of a dynamic e-Business will not necessarily see a need anymore to know the exact reason of performance degradations as long as a service provider is able to take appropriate remedies (or compensate its customers for the incurred service level violation).

Traditionally, end-to-end performance management has been the goal of traditional enterprise management efforts and is often explicitly listed as a requirement (see, e.g., [26]). However, the aforementioned privacy concerns of service providers and the service customers’ need for transparency make that an end-to-end view becomes unachievable (and irrelevant!) in a dynamic e-Business environment spanning multiple organizational domains.

Delegate Monitoring Tasks to Third Parties

Traditionally, an SLA is a bilateral agreement between a service customer and a service provider: The *enhanced Telecom Operations Map (eTOM)* [29], for example, defines various roles services providers can play; however, this work does not provide the delegation of management functionality to further service providers. We refer to the parties that establish and sign the SLA as **signatory parties**. In addition, SLA monitoring may require the involvement of third parties: They come into play when either a function needs

to be carried out that neither service provider nor customer wants to do, or if one signatory party does not trust the other to perform a function correctly. Third parties act then in a **supporting role** and are sponsored by either one or both signatory parties.

The targeted environment of our work is a typical service provider environment (Internet storefronts, B2B marketplaces, web hosting, ASP), which consists of multiple, independent parties that collaborate according to the terms and conditions specified in the SLA. Consequently, the services of our architecture are supposed to be distributed among the various parties and need to interact across organizational domains. Despite the focus on cross-organizational entities, WSLA can be applied to environments in which several (or even all of the) services reside within the boundaries of a single organizational domain, such as in a traditional corporate network.

The work of the IST Project FORM [8] is highly relevant for our work, since it focuses on SLAs in an inter-domain environment. FORM also deals with the important issue of federated accounting [3], which we do not address in this paper. An approach for a generic service model suitable for customer service management is presented in [9].

SLA-driven Resource Configuration

Since the terms and conditions of an SLA may entail setting configuration parameters on a potentially wide range of managed resources, an SLA management framework must accommodate the definition of SLAs that go beyond electronic/web services and relate to the supporting infrastructure. On the one hand, it needs to tie the SLA to the monitoring parameters exposed by the managed resources so that an SLA monitoring infrastructure is able to retrieve important metrics from the resources. [33] defines a MIB for SLA performance monitoring in an SNMP environment, whereas the SLA handbook from TeleManagement Forum [27] proposes guidelines for defining SLAs for telecom service providers.

An approach for the performance instrumentation of EJB-base distributed applications is described in [5]. The capability of mapping resource metrics to SLA parameters is crucial because a service provider must be able to answer the following questions before signing an SLA:

- Is it possible to accept an SLA for a specific service class given the fact that the capacity is limited?
- Can additional workload be accommodated?

On the other hand, it is desirable to derive configuration settings directly from SLAs. However, the heterogeneity and complexity of the management infrastructure makes configuration management a challenge. Successful work in this area often focuses on the network level: [10] describes a network configuration language; the Policy Core Information Model (PCIM) of the IETF

[23] provides a generic framework for defining policies to facilitate configuration management.

Existing work in the e-commerce area may be applied here as well since the concept of contract-driven configuration in e-commerce environments [11] and virtual enterprises [20, 13] has similarities to the SLA-driven configuration of managed resources.

WSLA Runtime Architecture

In this section, we describe the WSLA runtime architecture by breaking it down into its atomic building blocks, namely the elementary services needed to enable the management of an SLA throughout the phases of its lifecycle. The first part describes the information flows and interactions between the different services. The next section demonstrates how the SLA management services identified earlier cooperate in an inter-domain environment, where the task of SLA management itself is dynamically delegated to an arbitrary number of management service providers.

WSLA Services and their Interactions

The components described in this section are designed to address the “need to know” principle and constitute the atomic building blocks of the WSLA monitoring framework. The components are intended to interact across multiple domains; however, it is possible that some components may be co-located within a single domain and not necessarily exposed to objects residing within another domain.

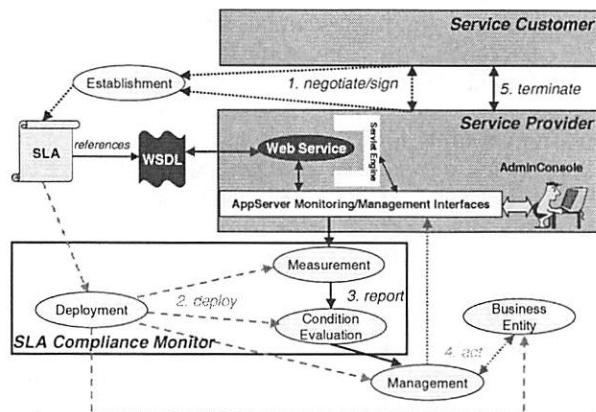


Figure 2: Interactions between the WSLA services.

Figure 2 gives an overview of the SLA management lifecycle, which consists of five distinct phases. We assume that an SLA is defined for a web service, which is running in the servlet engine of a web application server. The web application server exposes a variety of management information either through the graphical user interface of an administration console or at its monitoring and management interfaces, which are accessed by the various services of the WSLA framework.

The interface of the web service is defined by an XML document in the *Web Services Description*

Language (WSDL). The SLA references this WSDL document and extends the service definition with SLA management information. Typically, an SLA defines several SLA parameters, each referring to an operation of the web service. However, an SLA may also reference the service as a whole, or even compositions of multiple web services [30]. The phases and the services that implement the functionality needed during the various phases are as follows.

Phase 1: SLA Negotiation and Establishment

The SLA is being negotiated and signed by both signatory parties. This is done by means of an **SLA Establishment Service**, i.e., an SLA authoring tool that lets both signatory party establish, price and sign an SLA for a given service offering. This tool allows a customer to retrieve the metrics offered by a service provider, aggregate and combine them into various SLA parameters, request approval from both parties, define secondary parties and their tasks, and make the SLA document available for deployment to the involved parties (dotted arrows in Figure 2).

Phase 2: SLA Deployment

Deployment Service: The deployment service is responsible for checking the validity of the SLA and distributing it either in full or in appropriate parts to the involved components (dashed arrows in Figure 2). Since two signatory parties negotiate the SLA, they must inform the supporting parties about their respective roles and duties. Two issues must be addressed:

1. Signatory parties do not want to share the whole SLA with their supporting parties but restrict the information to the relevant information such that they can configure their components. Signatory parties must analyze the SLA and extract relevant information for each party. In the case of a measurement service, this is primarily the definition of SLA parameters and metrics. SLO evaluation services get the SLOs they need to verify. All parties need to know the definitions of the interfaces they must expose, as well as the interfaces of the partners they interact with.
2. Components of different parties cannot be assumed to be configurable in the same way, i.e., they may have heterogeneous configuration interfaces.

Thus, the deployment process contains two steps. In the first step, the SLA deployment system of a signatory party generates and sends configuration information in the *Service Deployment Information (SDI)* format (omitted for the sake of brevity), a subset of the language described later, to its supporting parties. In the second step, deployment systems of supporting parties configure their own implementations in a suitable way.

Phase 3: Measurement and Reporting

This phase deals with configuring the runtime system in order to meet one or a set of SLOs, and with

carrying out the computation of SLA parameters by retrieving resource metrics from the managed resources and executing the management functions (solid arrows in Figure 2). The following services implement the functionality needed during this phase:

Measurement Service: The Measurement Service maintains information on the current system configuration, and run-time information on the metrics that are part of the SLA. It measures SLA parameters such as availability or response time either from inside, by retrieving resource metrics directly from managed resources, or outside the service provider's domain, e.g., by probing or intercepting client invocations. A Measurement Service may measure all or a subset of the SLA parameters. Multiple measurement services may simultaneously measure the same metrics.

Condition Evaluation Service: This service is responsible for comparing measured SLA parameters against the thresholds defined in the SLA and notifying the management system. It obtains measured values of SLA parameters from the Measurement Service and tests them against the guarantees given in the SLA. This can be done each time a new value is available, or periodically.

Phase 4: Corrective Management Actions

Once the Condition Evaluation Service has determined that an SLO has been violated, corrective management actions need to be carried out. The functionality that needs to be provided in this phase spans two different services:

Management Service: Upon receipt of a notification, the management service (usually implemented as part of a traditional management platform) will retrieve the appropriate actions to correct the problem, as specified in the SLA. Before acting upon the managed system, it consults the business entity (see below) to verify if the proposed actions are allowable. After receiving approval, it applies the action(s) to the managed system.

It should be noted that the management component seeks approval for every proposed action from the business entity. The main purpose of the management service is to execute corrective actions on behalf of the managed environment if a Condition Evaluation Service discovers that a term of an SLA has been violated. While such corrective actions are limited today to opening a trouble ticket or sending an event to the provider's management system, we envision this component playing a crucial role in the future by acting as an automated mediator between the customer and provider, according to the terms of the SLA. This includes the submission of proposals to the management system of a service provider on how a performance problem could be resolved (e.g., proposing to assign a different traffic category to a customer if several categories have been defined in the SLA).

Our implementation addresses very simple corrective actions; finding a generic, flexible and

automatically executable mechanism for corrective management actions remains an open issue yet.

Business Entity: It embodies the business knowledge, goals and policies of a signatory party (here: service provider), which are usually kept confidential. Such knowledge enables the business entity to verify if the actions specified in the SLA (eventually some time ago) are still compatible with the actual business targets. If this is the case, the business entity will send a positive acknowledgement to the request of the Management Service; in case the proposed actions are in conflict with the actual goals of the service provider, its business entity will decline the request and the management service will refrain from carrying them out. It should be noted that declining prior agreed-upon actions may be regarded by another party as a breach of the SLA entailing, in an severe case, termination of the business relationship. Since it is unlikely that decisions of this importance will be left to the discretion of an automated system, we assume that the decision of the business entity requires human intervention. While we have implemented the aforementioned services, we have postponed an implementation of a business entity component until appropriate mechanisms for specifying and enforcing business policies are available.

Our experience shows that the tasks covered by these two services become extremely complicated as soon as sophisticated management actions need to be specified: First, a service provider would need to expose what management operations he is able to execute, which is very specific to the management platforms (products, architectures, protocols) he uses. Second, these management actions may become very complicated and may require human interaction (such as deploying new servers). Finally, due to the fact that the provider's managed resources are shared among various customers, management actions that satisfy an SLA with one customer are likely to impact the SLAs the provider has with other customers. The decision whether to satisfy the SLA (or deliberately break it) therefore is not a technical decision anymore, but rather a matter of the provider's business policies and, thus, lies beyond the scope of the work discussed in this paper. Consequently, only a few elements of the WSLA language address this phase of the service lifecycle.

Phase 5: SLA Termination

The SLA may specify the conditions under which it may be terminated or the penalties a party will incur by breaking one or more SLA clauses. Negotiations for terminating an SLA may be carried out between the parties in the same way as the SLA establishment is being done. Alternatively, an expiration date for the SLA may be specified in the SLA.

SLA Compliance Monitor Implementation

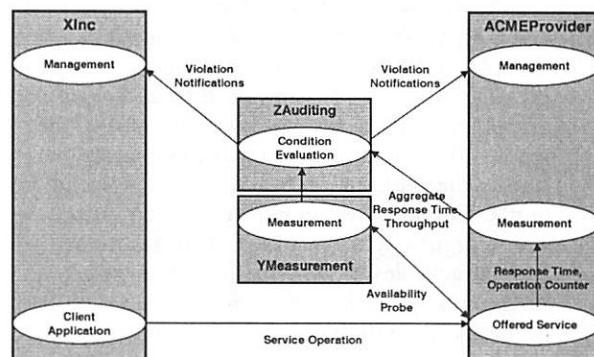
Figure 2 shows which WSLA services have been implemented. Because of their major importance and

their excellent suitability for automated processing, the **Deployment**, **Measurement** and **Condition Evaluation** services have been implemented by us. These services are implemented as Web Services themselves and are jointly referred to as **SLA Compliance Monitor**, which acts as a wrapper for the three services. For information where to download the implementation, the reader is referred to the 'Availability' section. Our ongoing implementation efforts, aimed at completing the WSLA framework, are described in 'Conclusions and Outlook.'

Signatory and Supporting Parties

Figure 3 gives an overview of a configuration where two signatory parties and two supporting parties collaborate in the monitoring of an SLA.

In bilateral SLAs, it is usually straightforward to define for each commitment who is the obliged and who is the beneficiary of the commitment. However, in an SLA containing more than two parties, it is not obvious which party guarantees what to whom. A clear definition of responsibilities is required. The WSLA environment involves multiple parties to enact an SLA instance. As mentioned above, a part of the monitoring and supervision activities can be assigned to parties other than the service provider and customer.



located in the provider's domain while another measurement service probes the service offered by the provider across the Internet from various locations. Keynote Systems, Inc. [15] is an example of such an external measurement service provider.

Despite the fact that a multitude of parties may be involved in providing a service, these interactions may be broken down into chained customer/provider relationships. Every interaction therefore involves only two roles, a sender and a recipient. During our work, we have not encountered a need for multi-party SLAs, i.e., SLAs that are *simultaneously* negotiated and signed by more than two parties. Multi-party contracts do not seem to provide enough value to justify their added complexity.

The WSLA Language

The WSLA language, specified in [19], defines a type system for the various SLA artifacts and is based on the XML Schema [34, 35]. We give an overview of the general structure of an SLA and motivate the various constructs of the WSLA language that will be described by means of examples in the subsequent sections: The information that needs to be processed by a Measurement Service is described later; then, we focus on the parts of the language a Condition Evaluation Service needs to understand for evaluating if a service level objective has been violated.

WSLA in a Nutshell

Figure 4 illustrates the typical elements of a SLA with signatory and supporting parties. Clearly, there are many variations of what types of information and which rules are to be included and, hence, enforced in a specific SLA.

The **Parties** section, consisting of the signatory parties and supporting parties fields identify all the contractual parties. **Signatory Party** descriptions contain the identification and the technical properties of a party, i.e., their interface definition and their addresses. The definitions of the **Supporting Parties** contain, in addition to the information contained in the signatory party descriptions, an attribute indicating the sponsor(s) of the party.

The **Service Description** section of the SLA specifies the characteristics of the service and its observable parameters as follows:

- For every **Service Operation**, one or more **Bindings**, i.e., the transport encoding for the messages to be exchanged, may be specified. Examples of such bindings are SOAP (Simple Object Access Protocol), MIME (Multipurpose Internet Mail Extensions) or HTTP (HyperText Transfer Protocol).
- In addition, one or more **SLA Parameters** of the service may be specified. Examples of such SLA parameters are *service availability*, *throughput*, or *response time*.
- As mentioned earlier, every SLA parameter refers to one (composite) **Metric**, which, in turn, aggregates one or more other (composite or resource) metrics, according to a measurement directive or a function. Examples of composite metrics are *maximum response time of a service*, *average availability of a service*, or *minimum throughput of a service*. Examples of resource metrics are: *system uptime*, *service outage period*, *number of service invocations*.
- **Measurement Directives** specify how an individual metric can be accessed. Typical

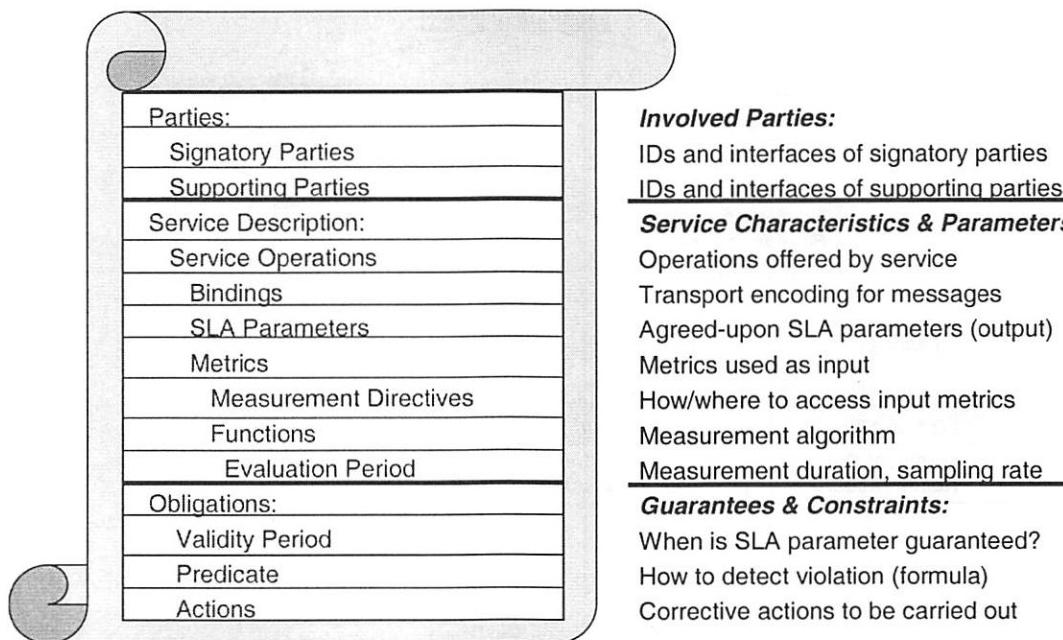


Figure 4: General structure of an SLA.

examples of measurement directives are the uniform resource identifier of a hosted computer program, a protocol message, or the command for invoking scripts or compiled programs.

- **Functions** are the measurement algorithm, or formula, that specifies how a composite metric is computed. Examples of functions are formulas of arbitrary length containing average, sum, minimum, maximum, and various other arithmetic operators, or time series constructors.
- For every function, an **Evaluation Period** is specified. It defines the time intervals during which the functions are executed to compute the metrics. These time intervals are specified by means of *start time*, *duration*, and *frequency*. Examples of the latter are *weekly*, *daily*, *hourly*, or *every minute*.

Obligations, the last section of an SLA, define various guarantees and constraints that may be imposed on the SLA parameters:

- First, the **Validity Period** is specified; it indicates the time intervals for which a given SLA parameter is valid, i.e., when the SLO may be applied. Examples of validity periods are *business days*, *regular working hours* or *maintenance periods*.
- The **Predicate** specifies the threshold and the comparison operator (greater than, equal, less than, etc.) against which a computed SLA parameter is to be compared. The result of the predicate is either *true* or *false*.

• **Actions**, finally, are triggered whenever a predicate evaluates to *true*, i.e., a violation of an SLO has occurred. Actions are e.g., *sending an event to one or more signatory and supporting parties*, *opening a trouble ticket or problem report*, *payment of penalty*, or *payment of premium*. Note that, as stated in the latter case, a service provider may very well receive additional compensation from a customer for exceeding an obligation, i.e., obligations reflect constraints that may trigger the payment of credits from any signatory party to another signatory or supporting party. Also note that zero or more actions may be specified for every SLA parameter.

Service Description: Associating SLA Parameters with a Service

The purpose of the service description is the clarification of three issues: *To which service do SLA parameters relate?* *What are the SLA parameters?* *How are the SLA parameters measured or computed?* This is the information a Measurement Service requires to carry out its tasks. A sample service description is depicted in Figure 5.

Service Objects and Operations

The service object, depicted at the top of Figure 5, provides an abstraction for all conceptual elements for which SLA parameters and the corresponding metrics can be defined. In the context of Web Services, the most detailed concept whose quality aspect can be described separately is the individual operation (in a binding) described in a WSDL specification. In our

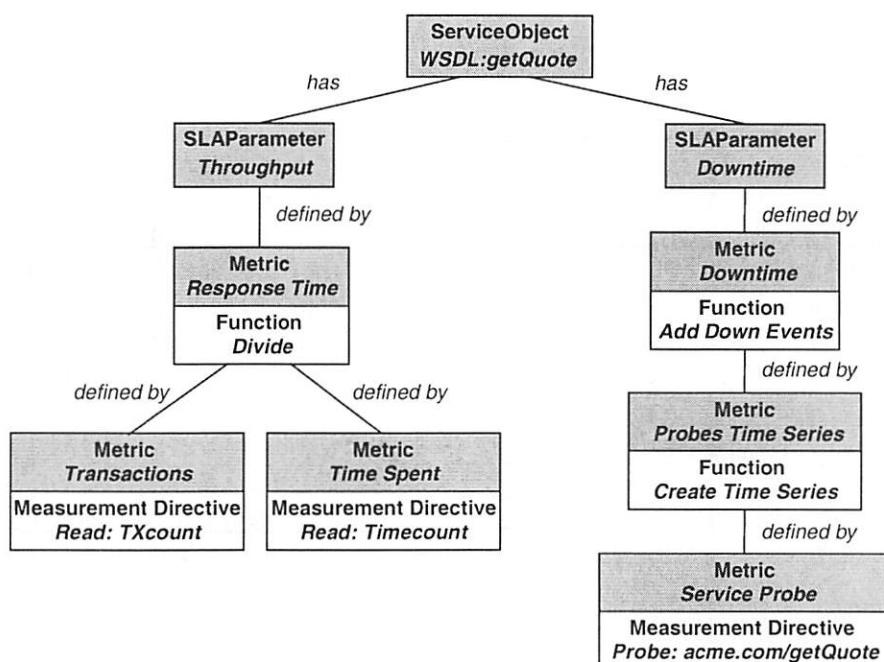


Figure 5: Sample elements of a service description.

example, the operation `getQuote` is the service object. In addition, quality properties of groups of WSDL operations can be defined – the operation group being the service object in this case. Outside the scope of Web Services, business processes, or parts thereof, can be service objects (e.g., defined in WSFL [18]). Service objects have a set of SLA parameters, a set of metrics that describe how SLA parameters are computed or measured and a reference to the service itself that is the subject of the service object abstraction.

While the format for SLA parameters and metrics is the same for all services (though not their individual content), the reference to the service depends on the particular way in which the service is described. For example, service objects may contain references to operations in a WSDL file.

SLA Parameters and Metrics

SLA parameters are properties of a service object; each SLA parameter has a name, type and unit. SLA parameters are computed from metrics, which either define how a value is to be computed from other metrics or describe how it is measured. For this purpose, a metric either defines a function that can use other metrics as operands or it has a measurement directive that describes how the metric's value should be measured. Since SLA parameters are the entities that are surfaced by a Measurement Service to a Condition Evaluation Service, it is important to define which party is supposed to provide the value (Source) and which parties can receive it, either event-driven (Push) or through polling (Pull). Note that one of our design choices is that SLA parameters are *always* the result of a computation, i.e., no SLA parameters can be defined as input parameters for computing other SLA parameters. In Figure 5, one metric is retrieved by probing an interface (Service Probe) while the other

ones (TXcount, Timecount) are directly retrieved from the service provider's management system.

```
<SLAParameter name="UpTimeRatio"
  type="float" unit="downEvents/hour">
  <Metric>UpTimeRatioMetric</Metric>
  <Communication>
    <Source>ACMEProvider</Source>
    <Pull>ZAuditing</Pull>
    <Push>ZAuditing</Push>
  </Communication>
</SLAParameter>
```

Figure 6: Defining an SLA Parameter UpTimeRatio.

Figure 6 depicts how an SLA parameter `UpTimeRatio` is defined. It is assigned the metric `UpTimeRatioMetric`, which is defined independently of the SLA parameter for being used potentially multiple times. `ACMEProvider` promises to send (push) new values to `ZAuditing`, which is also allowed to retrieve new values on its own initiative (pull). The purpose of a metric is to define how to measure or compute a value. Besides a name, a type and a unit, it contains either a function or a measurement directive and a definition of the party that is in charge of computing this value.

Figure 7 shows an example composite metric containing a function. `UpTimeRatioMetric` is of type double and has no unit. `YMeasurement` is in charge of computing this value. The example illustrates the concept of a function: The number of occurrences of "0" in a time series of the metric `StatusTimeSeries` – assuming this represents a down event in time series of probes once per minute – is divided by 1440 (the number of minutes of a day) to yield the downtime ratio. This value is subtracted from 1 to obtain the `UpTimeRatio`. Specific functions, such as `Minus`, `Plus` or `ValueOccurs` are extensions of the common function type.

```
<Metric name="UpTimeRatioMetric" type="double" unit="">
  <Source>YMeasurement</Source>
  <Function xsi:type="Minus" resultType="double">
    <Operand>
      <LongScalar>1</LongScalar>
    </Operand>
    <Operand>
      <Function xsi:type="Divide" resultType="long">
        <Operand>
          <Function xsi:type="ValueOccurs" resultType="long">
            <Metric>StatusTimeSeries</Metric>
            <Value>
              <LongScalar>0</LongScalar>
            </Value>
          </Function>
        </Operand>
        <Operand>
          <LongScalar>1440</LongScalar>
        </Operand>
      </Function>
    </Operand>
  </Function>
</Metric>
```

Figure 7: Defining a Metric UpTimeRatioMetric.

Operands of functions can be metrics, scalars and other functions. It is expected that a measurement service, provided either by a signatory or a supporting party, is able to compute functions. Specific functions can be added to the language as needed.

A **Measurement Directive**, depicted in Figure 8, specifies *how* the metric is retrieved from the source (either by means of a well-defined query interface offered by the Service Provider, or directly from the instrumentation of a managed resource by means of a management protocol operation). A specific type of measurement directive is used in the example above: StatusRequest. It contains a URL that is used for probing whether the getQuote operation is available. Apparently, other ways to measure values require an entirely different set of information items, e.g., an SNMP port, an object identifier (OID) and an instance identifier to retrieve a counter.

Obligations: SLOs and Action Guarantees

Based on the common ontology established in the service definition part of the SLA, the parties can unambiguously define the respective guarantees that they give each other. The WSLA language provides two types of obligations:

- **Service level objectives** represent promises with respect to the state of SLA parameters.
- **Action guarantees** are promises of a signatory party to perform an action. This may include notifications of service level objective violations or invocation of management operations.

Important for both types of obligations is the definition of the obliged party and the definition of when the obligations need to be evaluated. Both have a similar syntactical structure (as previously depicted in Figure 4). However, their semantics are different. The content of an obligation is refined in a service level objective or an action guarantee.

Service Level Objectives

A service level objective expresses a commitment to maintain a particular state of the service in a given period. Any party can take the obliged part of this guarantee; however, this is typically the service provider.

A service level objective has the following elements: Obliged is the name of a party that is in charge of delivering what is promised in this guarantee. One or many ValidityPeriods define when the SLO is applicable.

A logic Expression defines the actual content of the guarantee, i.e., what is asserted by the service provider to the service customer. Expressions follow first order logic and contain the usual operators *and*,

or, *not*, etc., which connect either predicates or, again, expressions. Predicates can have SLA parameters and scalar values as parameters. By extending an abstract predicate type, new domain-specific predicates can be introduced as needed. Similarly, expressions could be extended, e.g., to contain variables and quantifiers. This provides the expressiveness to define complex states of the service.

A service level objective may also have an EvaluationEvent, which defines when the expression of the service level objective should be evaluated. The most common evaluation event is NewValue, each time a new value for an SLA parameter used in a predicate is available. Alternatively, the expression may be evaluated according to a Schedule. A schedule is a sequence of regularly occurring events. It can be defined within a guarantee or may refer to a commonly used schedule.

The example in Figure 9 illustrates a service level objective given by ACMEProvider and valid for a full month in the year 2001. It guarantees that the SLA parameter ThroughPutRatio must be greater than 1000 if the SLA parameter UpTimeRatio is less than 0.9, i.e., the ThroughPutRatio must be above 1000 transactions per minute even if the overall availability is below 90%. This condition should be evaluated each time a new value for the SLA parameter is available. Note that we deliberately chose that validity periods are always specified with respect to a single SLA parameter, and thus only indirectly applicable to the scope of the overall SLA. Alternatively, validity periods to the overall SLA (possibly in addition to the validity periods for each SLA parameter) could be possible, but we found that this granularity is too coarse.

Action Guarantees

An action guarantee expresses a commitment to perform a particular activity if a given precondition is met. Any party can be the obliged of this kind of guarantee. This particularly includes also the supporting parties of the SLA.

An action guarantee comprises the following elements and attributes: Obliged is the name of a party that must perform an action as defined in this guarantee. A logic Expression defines the precondition of the action. The format of this expression is the same as the format of expression in service level objectives. An important predicate for action guarantees is the Violation predicate that determines whether another guarantee, in particular a service level objective, has been violated. An EvaluationEvent or an evaluation Schedule defines when the precondition is evaluated.

```
<Metric name="ServiceProbe" type="integer" unit="">
  <Source>YMeasurement</Source>
  <MeasurementDirective xsi:type="StatusRequest" resultType="integer">
    <RequestURL>http://ymeasurement.com/StatusRequest/GetQuote</RequestURL>
  </MeasurementDirective>
</Metric>
```

Figure 8: Defining a Measurement Directive StatusRequest.

`QualifiedAction` contains a definition of the action to be invoked at a particular party. The concept of a qualified action definition is similar to the invocation of an object method in a programming language, replacing the object name with a party name. The party of the qualified action can be the obliged or another party. The action must be defined in the corresponding party specification. In addition, the specification of the action includes the marshalling of its parameters. One or more qualified actions can be part of an action guarantee.

`ExecutionModality` is an additional means to control the execution of the action. It can be defined whether the

action should be executed if a particular evaluation of the expression yields true. The purpose is to reduce, for example, the execution of a notification action to a necessary level if the associated expression is evaluated very frequently. Execution modality can be either: *always*, *on entering a condition* or *on entering and leaving a condition*. The example depicted in Figure 10 illustrates an action guarantee.

In the example, `ZAuditing` is obliged to invoke the notification action of the service customer `XInc` if a violation of the service level objective `SLO_For_ThroughPut_and_UpTime` (cf. Figure 9) occurs. The precondition should be evaluated every time the

```
<ServiceLevelObjective name="SLO_For_ThroughPut_and_UpTime">
  <Obliged>ACMEProvider</Obliged>
  <Validity>
    <Start>2001-11-30T14:00:00.000-05:00</Start>
    <End>2001-12-31T14:00:00.000-05:00</End>
  </Validity>
  <Expression>
    <Implies>
      <Expression>
        <Predicate xsi:type="Less">
          <SLAParameter>UpTimeRatio</SLAParameter>
          <Value>0.9</Value>
        </Predicate>
      </Expression>
      <Expression>
        <Predicate xsi:type="Greater">
          <SLAParameter>ThroughPutRatio</SLAParameter>
          <Value>1000</Value>
        </Predicate>
      </Expression>
    </Implies>
  </Expression>
  <EvaluationEvent>NewValue</EvaluationEvent>
</ServiceLevelObjective>
```

Figure 9: Defining a Service Level Objective `SLO_For_ThroughPut_and_UpTime`.

```
<ActionGuarantee name="Must_Send_Notification_Guarantee">
  <Obliged>ZAuditing</Obliged>
  <Expression>
    <Predicate xsi:type="Violation">
      <ServiceLevelObjective>
        SLO_For_ThroughPut_and_UpTime
      </ServiceLevelObjective>
    </Predicate>
  </Expression>
  <EvaluationEvent>NewValue</EvaluationEvent>
  <QualifiedAction>
    <Party>XInc</Party>
    <Action actionPerformed="notification" xsi:type="Notification">
      <NotificationType>Violation</NotificationType>
      <CausingGuarantee>
        Must_Send_Notification_Guarantee
      </CausingGuarantee>
      <SLAParameter>ThroughPutRatio UpTimeRatio</SLAParameter>
    </Action>
  </QualifiedAction>
  <ExecutionModality>Always</ExecutionModality>
</ActionGuarantee>
```

Figure 10: Defining an ActionGuarantee `Must_Send_Notification_Guarantee`.

evaluation of the SLO Must_Send_Notification_Guarantee returns a new value. The action has three parameters: the type of notification, the guarantee that caused it to be sent, and the SLA parameters relevant for understanding the reason of the notification. The notification should always be executed.

Conclusions and Outlook

This paper has introduced the novel WSLA framework for electronic services, in particular Web Services. The WSLA language allows a service provider and its customer to define the quality of service aspects of the service. The concept of supporting parties allows signatory parties to include third parties into the process of measuring the SLA parameters and monitoring the obligations associated with them. In order to avoid the potential ambiguity of high-level SLA parameters, parties can define precisely how resource metrics are measured and how composite metrics are computed from others. The WSLA language is extensible and allows us to derive new domain-specific or technology-specific elements from existing language elements. The explicit representation of service level objectives and action guarantees provides a very flexible mechanism to define obligations on a case-by-case basis. Finally, the detachment from the service description itself makes the WSLA language and its associated services applicable to a wide range of electronic services.

We have developed a prototype that implements a total of three different WSLA services: First, a deployment service to provide the measurement and condition evaluation services with the SLA elements they need to know; second, a measurement service that can interpret measurement directives for the instrumentation of a Web services gateway and can aggregate high-level metrics using a rich set of functions for arithmetic and time series. Third, a general-purpose condition evaluation service has been implemented that supports a wide range of predicates. Currently, we provide extensions to the WSLA language that apply to quality aspects of business processes and a template format for advertising SLAs in service registries such as UDDI. In addition, we are working on an SLA editing environment. The integration with existing resource management systems and architectures is currently underway, with a special focus on the *Common Information Model (CIM)*.

Availability

The SLA Compliance Monitor is included in the current version 3.2 of the IBM Web Services Toolkit and can be downloaded from <http://www.alphaworks.ibm.com/tech/webservicestoolkit>.

Acknowledgments

The authors would like to express their gratitude to Asit Dan, Richard Franck and Richard P. King for their contribution. The authors are also indebted to

Steve Traugott of TerraLuna, LLC. for his constructive suggestions for improving the quality of this paper.

Biography

Alexander Keller is a Research Staff Member at the IBM Thomas J. Watson Research Center in Yorktown Heights, NY, USA. He received his M.Sc. and a Ph.D. in Computer Science from Technische Universität München, Germany, in 1994 and 1998, respectively and has published more than 30 refereed papers in the area of distributed systems management. He does research on service and application management, information modeling for e-business systems, and service level agreements. He is a member of GI, IEEE and the DMTF CIM Applications Working Group.

Heiko Ludwig is a visiting scientist at the IBM Thomas J. Watson Research Center since June 2001. As a member of the Distributed Systems and Services department he works in the field of electronic contracts, both contract representation and architectures for contract-based systems. He holds a Master's degree (1992) and a Ph.D. (1997) in computer science and business administration from Otto-Friedrich University Bamberg, Germany.

References

- [1] ASP Industry Consortium, *White Paper on Service Level Agreements*, 2000.
- [2] Bhoj, P., S. Singhal, and S. Chutani, "SLA Management in Federated Environments," *Proceedings of the Sixth IFIP/IEEE Symposium on Integrated Network Management (IM'99)*, Boston, MA, IEEE Publishing, pp. 293-308, May 1999.
- [3] Bhushan, B., M. Tschichholz, E. Leray, and W. Donnelly, "Federated Accounting: Service Charging and Billing in a Business-To-Business Environment," *Proceedings of the 7th IFIP/IEEE International Symposium on Integrated Network Management*, (N. Anerousis, G. Pavlou, and A. Liotta, editors), Seattle, WA, USA, IEEE Publishing, pp. 107-121, May 2001.
- [4] Dan, A., D. Dias, R. Kearney, T. Lau, T. Nguyen, F. Parr, M. Sachs, and H. Shaikh, "Business-to-Business Integration with tpaML and a B2B Protocol Framework," *IBM Systems Journal*, Vol. 40, No. 1, February 2001.
- [5] Debusmann, M., M. Schmidt, and R. Kroeger, "Generic Performance Instrumentation of EJB Applications for Service Level Management," Stadler and Ulema [28].
- [6] ebXML – *Creating a Single Global Electronic Market*, <http://www.ebxml.org>.
- [7] ebXML Trading-Partners Team, *Collaboration-Protocol Profile and Agreement Specification, Version 1.0*, UN/CEFACT and OASIS, May 2001.
- [8] FORM Consortium, *Final Inter-Enterprise Management System Model*, Deliverable 11, IST

- Project FORM: Engineering a Co-operative Inter-Enterprise Framework Supporting Dynamic Federated Organisations Management, <http://www.ist-form.org>, February 2002.
- [9] Garschhammer, M., R. Hauck, H.-G. Hegering, B. Kempfer, M. Langer, M. Nerb, I. Radisic, H. Roelle, and H. Schmidt, "Towards Generic Service Management Concepts: A Service Model Based Approach," *Proceedings of the 7th IFIP/IEEE International Symposium on Integrated Network Management*, pp. 719-732.
 - [10] Gopal, R., "Unifying Network Configuration and Service Assurance with a Service Modeling Language," In Stadler and Ulema [28], pp. 711-725.
 - [11] Griffel, F., M. Boger, H. Weinreich, W. Lamersdorf, and M. Merz, "Electronic contracting with COSMOS – How to establish, Negotiate and Execute Electronic Contracts on the Internet," *Proceedings of the Second International Enterprise Distributed Object Computing Workshop (EDOC '98)*, La Jolla, CA, USA, October 1998.
 - [12] Grosof, B. and Y. Labrou, "An Approach to using XML and a Rule-based Content Language with an Agent Communication Language," *Proceedings of the IJCAI-99 Workshop on Agent Communication Languages (ACL-99)*, 1999.
 - [13] Hoffner, Y., S. Field, P. Grefen, and H. Ludwig, "Contract-driven Creation and Operation of Virtual Enterprises," *Computer Networks*, Vol. 37, pp. 111-136, 2001.
 - [14] Keller, A., G. Kar, H. Ludwig, A. Dan, and J. L. Hellerstein, "Managing Dynamic Services: A Contract based Approach to a Conceptual Architecture," In Stadler and Ulema [28], pp. 513-528.
 - [15] Keynote – The Internet Performance Authority, <http://www.keynote.com>.
 - [16] Kreger, H., *Web Services Conceptual Architecture 1.0*, IBM Software Group, May 2001.
 - [17] Lewis, L., *Managing Business and Service Networks*, Kluwer Academic Publishers, 2001.
 - [18] Leymann, F., *Web Services Flow Language (WSFL) 1.0*, IBM Software Group, May 2001.
 - [19] Ludwig, H., A. Dan, R. Franck, A. Keller, and R. P. King, *Web Service Level Agreement (WSLA) Language Specification*, IBM Corporation, July 2002.
 - [20] Ludwig, H. and Y. Hoffner, "The Role of Contract and Component Semantics in Dynamic E-Contract Enactment Configuration," *Proceedings of the 9th IFIP Workshop on Data Semantics (DS9)*, pp. 26-40, 2001.
 - [21] McCloghrie, K., D. Perkins, and J. Schoenwaelder, *Structure of Management Information – Version 2 (SMIV2)*, RFC 2578, IETF, April 1999.
 - [22] Merz, M., F. Griffel, T. Tu, S. Müller-Wilken, H. Weinreich, M. Boger, and W. Lamersdorf, "Supporting Electronic Commerce Transactions with contracting Services," *International Journal of Cooperative Information Systems*, Vol. 7, No. 4, pp. 249-274, 1998.
 - [23] Moore, B., E. Ellesson, J. Strassner, and A. Westerinen, *Policy Core Information Model – Version 1 Specification*, RFC 3060, IETF, February 2001.
 - [24] Rodosek, G. Dreö and L. Lewis, "Dynamic Service Provisioning: A User-Centric Approach," *Proceedings of the 12th Annual IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM 2001)*, O. Festor and A. Pras, editors, Nancy, France, IFIP/IEEE, INRIA Press, pp. 37-48, October 2001.
 - [25] Schopp, B., A. Runge, and K. Stanoevska-Slabeva, "The Management of Business Transactions through Electronic Contracts," *Proceedings for the Tenth International Workshop on Database and Expert Systems Applications*, A. Camelli, A. Min Tjoa, and R. R. Wagner, editors, Florence, Italy, IEEE Computer Society Press, pp. 824-831, 1999.
 - [26] SLA and QoS Management Team, *Service Provider to Customer Performance Reporting: Information Agreement*, Member Draft Version 1.5 TMF 602, TeleManagement Forum, June 1999.
 - [27] SLA Management Team, *SLA Management Handbook*, Public Evaluation Version 1.5 GB 917, TeleManagement Forum, June 2001.
 - [28] Stadler, R. and M. Ulema, editors, *Proceedings of the IEEE/IFIP Network Operations and Management Symposium*, Florence, Italy, IEEE Press, April 2002.
 - [29] Enhanced Telecom Operations Map: The Business Process Framework, Member Evaluation Version 2.7 GB 921, TeleManagement Forum, April 2002.
 - [30] Tosic, V., B. Pagurek, B. Esfandiari, and K. Patel, "Management of Compositions of E- and M-Business Web Services with multiple Classes of Service," Stadler and Ulema [28], pp. 935-937.
 - [31] UDDI Version 2.0 API Specification, Universal Description, Discovery and Integration, <http://uddi.org>, June 2001.
 - [32] Verma, D., *Supporting Service Level Agreements on IP Networks*, Macmillan Technical Publishing, 1999.
 - [33] White, K., *Definition of Managed Objects for Service Level Agreements Performance Monitoring*, RFC 2758, IETF, February 2000.
 - [34] XML Schema Part 1: Structures, W3C Recommendation, W3 Consortium, May 2001.
 - [35] XML Schema Part 2: Datatypes, W3C Recommendation, W3 Consortium, May 2001.

HotSwap – Transparent Server Failover for Linux

Noel Burton-Krahn – HotSwap Network Solutions

ABSTRACT

HotSwap is a program that provides transparent failover for existing UNIX servers without modification or special hardware. HotSwap runs two instances of a server on independent machines in sync, so that if either machine fails, the other may assume control without breaking TCP connections or losing application state. Replication and failover is transparent to both clients and servers. Servers are not aware that a backup replica is maintaining state. Clients are unaware that a backup server has taken over from a failed master. This system is applicable to a wide variety of common servers including Java, Apache, and PostgreSQL and other servers that may have no other mechanisms for fault tolerance.

Introduction

Internet server applications must be scalable to many users, constantly available, and provide reliable service despite server failures, maintenance interruptions, and disasters. These features are critical in the long term, but are usually only considered after initial development. Most server applications today are developed with inexpensive components that do not support reliability, high availability or scalability, or any form of fault tolerance.

Adding fault tolerance to an existing system can be difficult and expensive, and may not be possible for some kinds of server applications. Many Internet server applications are developed using freely available tools like Apache, PHP, MySQL, etc. None of these applications has built-in fault tolerance. Current techniques for adding fault tolerance will allow clients to reconnect to a new server if one fails, but connections and state at the failed server will be lost. Fault-tolerant database and shared file servers are expensive. Implementing fault tolerance in a custom server is difficult.

HotSwap is a program that adds transparent fault tolerance to existing servers without modification. Application state is duplicated on two independent boxes that run in parallel. If one fails the other can continue without interrupting client connections. Replication is transparent to clients and servers, adding transparent failover to existing servers without modification.

There are several other techniques for fault tolerance [Coulouris01]. Each makes tradeoffs between degree of fault coverage, cost, and abilities. Scalability is the ability to increase performance by adding system components. Availability is the probability that a system is functioning properly at any moment in time. Reliability is the probability that a system will continue to function for a fixed period of time. Recoverability is the ability of a system to return to a functioning state without data loss after a failure. Total cost includes hardware, software, development, training, and maintenance.

Disaster recovery is the ability to recover from a large-scale disaster like fire or earthquake that can damage a wide area. A single system image reduces maintenance costs when several components can be maintained as a single logical unit.

Different capabilities compete with each other. Adding system components to increase scalability and availability will increase cost and may decrease reliability. Reliability and recoverability require synchronized backups, which may decrease performance and scalability. Disaster recovery requires backups separated by long distances at reduced bandwidth, slowing system synchronization and performance.

Faults may come from software bugs, hardware failures, or disasters. Hardware failures should be masked by switching to a backup system. Disaster may strike a whole building or geographical area. Recovering data after a disaster is crucial, but restoring network connections may require new routing.

Different servers have different requirements for fault tolerance. Web servers use short transactions, which can usually be repeated if they fail. Databases and file servers that update client data must be careful to maintain consistency with their backups. Teleconferencing and gaming servers maintain real-time state in memory.

Let's quickly review current techniques for adding fault tolerance.¹

Periodic Backup

Periodic backup is the easiest way to add disaster recovery. If a system crashes, a new one must be rebuilt from backup. Starting a new database server and restoring its backup may take some time. Changes to the system since the last backup are lost. Running applications may have to be shut down to prevent file

¹The Aberdeen group has published an excellent comparison of current high-availability products for Linux at <http://www.legato.com/resources/whitepapers/Aberdeen%20White%20Paper%20-%20Linux1.pdf>.

updates during backup. Connections to the system will be lost if it fails, and will have to be restarted when the new system is restored.

There are backup schemes for whole processes, not just files. Process checkpointing and hijacking [Skoglund00] is a technique of preserving and replicating application state by serializing the entire state at some point and restoring it later. A process checkpoint consists of its data pages, executable path, and system descriptor like open files, file pointers, etc. Condor² [Zandy99], [Litzkow97] uses application checkpointing to move an application completely from one machine to another. Like backing up data files, checkpointing is not suitable for real-time synchronization between two running processes.

Server Clusters

A server cluster uses a front-end director to distribute client requests over several back-end servers. All back-end servers provide a consistent application interface. Consistency requires that all back-end servers use a shared database or file system. If a back-end server fails, the director will send new client requests to another server in the pool. The new server will retrieve the client's state from the shared database.

The Linux Virtual Server³ project provides a director and a framework for back-end servers to use a shared Coda file system. F5Networks's BigIP⁴ acts as a Director monitoring RealServer health and distributing requests. BigIP also works redundantly and preserves client connections and encryption state on failures. Cisco's LocalDirector⁵ is a similar product.

Directors are usually redundant. If one fails, the other assumes its IP address to accept new connections. Directors monitor the health of the back-end servers and remove failed servers from the pool. Directors may terminate client connections and attempt to repeat client requests if a back-end server fails. Directors do not address reliability or recoverability. That's up to the back-end servers and common database.

²<http://www.cs.wisc.edu/condor/>.

³<http://www.linuxvirtualserver.org/>.

⁴<http://www.f5networks.com/>.

⁵<http://www.cisco.com/warp/public/cc/pd/cxsr/400/index.shtml>.

It is important to distinguish between availability and reliability in a server cluster. Clusters increase availability and scalability, but not reliability. If a server fails, another may be substituted for new connections, and thus availability is increased. However, once a client is connected to a back-end server, the connection's reliability is determined by the reliability of the chain of the director, back-end-server, and common database. The more components in the chain, the less reliable it is. Consider a server failure part way through a long file download. The replacement server will not know the state of the connection at the time of the failure, and the download must be restarted from the beginning.

Some directors like NetScaler⁶ buffer the client transaction requests and will repeat them if the server fails. These are built to support short non-interactive transactions, like HTTP requests. Interactive or unbounded connections cannot be buffered and retried at the director level.

Server clusters are ideal for web server applications. There are many simultaneous client connections, but they are each short and mostly read-only. The client's session (e.g., a shopping cart) is the only information that changes frequently, and that is stored on a shared database. None of the servers are expected to maintain state themselves. Client connections will be broken if a Director or RealServer fails. A broken connection is a minor inconvenience unless it's during a long download that must be repeated.

It is up to the back-end servers to synchronize shared state. Server machines may all connect to a central database or file server. If a middle server fails, all its state is lost and client connections are broken, but new client connections are directed to a new server. If a shared database fails, the whole cluster fails.

Fault Tolerant Applications

Some commercial database and file servers have built-in support for system synchronization and fail-over. Strict application consistency can seriously degrade performance. Lazy replication improves performance but relaxes consistency guarantees. Database

⁶<http://www.netscaler.com/product/technology.html>.

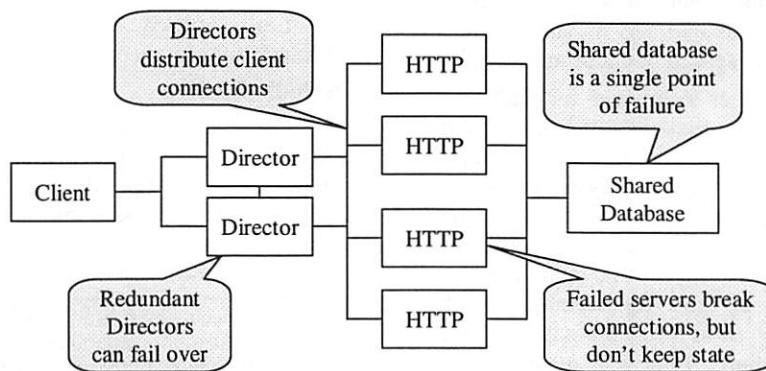


Figure 1: Schematic diagram of server cluster.

replication is still an active area of research [Patiño00, Wiesmann00].

Commercial databases like Oracle⁷ and Solid⁸ provide good application-level fault tolerance. Fault tolerant file servers include NetApp Filer.⁹

Server clusters need a fault-tolerant shared database for reliability. However, most fault-tolerant applications are expensive. Rewriting an application to use a new database is not trivial. Converting an existing application usually requires considerable expense, training, and maintenance.

Fault-Tolerant Programming Frameworks

Developing a fault tolerant server is difficult. Another option is to write or rewrite a custom server from scratch using a development framework that supports fault tolerance like J2EE and Enterprise Java Beans (EJB¹⁰)

These frameworks work well for the domain they were designed for. EJB is designed for writing Web-like applications where scripts provide an interface to a common database. EJB provides facilities for load balancing requests, maintaining client sessions, and fault tolerance.

Close examination of the EJB specification however reveals a limitation of EJB fault tolerance: transactions must be idempotent. If a transaction fails, the framework will automatically repeat it. Repeating a transaction must be equivalent to doing it exactly once. Fetching a record from a database is idempotent, but decrementing a balance is not. Applications that require non-idempotent operations have to implement their own fault tolerance.

Fault-Tolerant Hardware

Fault tolerant hardware has good performance, but it has been traditionally very expensive. Lately though, hosts with all redundant hardware components have become available at competitive prices. The Stratus ftServer¹¹ is quite reasonable.

Other systems use shared hardware like a shared SCSI disk to preserve state when a master fails. The shared disk itself may be fault-tolerant. However, the server's memory state and all client connections will be lost.

These are excellent solutions for avoiding hardware failures. They are often expensive though. Backup servers are connected by cables, so they are cannot be geographically separated. When disaster strikes the backup and master will both be vulnerable.

The different levels of RAID illustrate the trade-offs between redundancy, performance, cost, and

recovery. The simplest, RAID1, mirrors two disks. It is completely redundant, and does not increase performance. Higher RAID levels trade scalability for redundancy at increased cost.

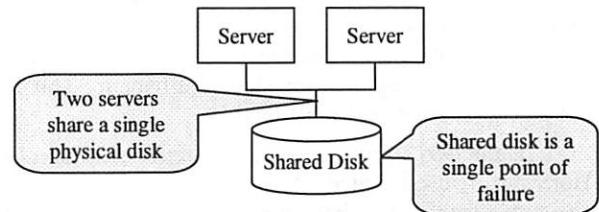


Figure 2: Shared hardware with a single point of failure.

TCP Connection Migration

All the servers mentioned so far break client connections on failure because they do not attempt to preserve TCP state. TCP connections are managed by the operating system itself. Even if an application synchronizes its internal state with a backup, the backup will not be able to reconstruct the sequence numbers, windows sizes, and timeouts of the master's TCP stack because that information is in the kernel, not the application.

There is recent research on moving TCP state to another machine. However, this research does not address how a client or server should transfer application state as well as TCP state. These systems provide facilities for moving TCP connections from a failed host to another. However, the server on the new host is responsible for ensuring that its state is consistent with the failed server. The server must be explicitly written to synchronize application state with a backup server.

[Snoeren01] describes extensions to the TCP protocol that allow a server or client to redirect a TCP connection to a new server without breaking the connection. This is handy for load balancing and avoiding failed servers. However, the TCP stack at both ends must be written to recognize these new TCP packet options. Replacement servers must implement their own synchronization for application state.

The reliable sockets system, Rocks [Zandy01], can preserve and reconnect TCP connections after link failures, address changes, and extended disconnection. Rocks does work without recompiling programs. However, servers must be rewritten to synchronize state. Rocks provides an API for managing and detecting resumed TCP connections.

Some redundant server cluster directors can synchronize both TCP and SSL state with their backups. If one director fails, the backup can continue the TCP connections. This only applies to the directors, not the back-end servers. A back-end server's TCP state is lost on failure.

[Alvisi00] proposes a system that allows a server to keep its TCP connections open until it restarts after

⁷www.oracle.com.

⁸www.solidtech.com.

⁹<http://www.netapp.com/>.

¹⁰<http://java.sun.com/products/ejb/>.

¹¹<http://www.stratus.com/products/nt/>.

failure. If the application is written to exploit this feature and it can reconstruct its state after failure, it can avoid closing client connections. However, the server must still be able to reconstruct its state after a crash.

[Aghdaie01] Presents a web server that preserves TCP connections and server state over failures. This is an example of a web server specially written to transfer both TCP and application state to a backup. This is not a general solution for all servers.

[Daniel99] presented a system similar to HotSwap. It used ptrace() to catch, redirect, and synchronize system calls between servers. However, that system used a central modified NFS server to provide a common file system. HotSwap does not use a shared file system; all servers are totally independent.

HotSwap

HotSwap maximizes availability and reliability by providing a hot backup server that maintains a complete independent copy of a master server's state. The backup is complete and dynamic, so it can take over all client connections when a server fails without interruption. It minimizes cost by adding this ability to almost any server without modification or special hardware. Both master and backup appear as a single computer on the network, and thus HotSwap provides a single system image. The tradeoff is a small amount of overhead to keep the master and backup servers in sync. HotSwap does not address scalability; backup servers do not share the load.

How it Works

HotSwap starts two identical instances of the same set of programs on two independent machines, a master and a backup. The programs are started from the same initial state, with duplicate file systems. As they run, HotSwap ensures that both copies are synchronized.

Synchronization means that both the master and backup programs see exactly the same input and produce exactly the same output. When a client connects, both servers receive the new connection. When a client sends data, both servers receive it. When the master server makes a system call, like requesting the current time, HotSwap ensures the backup gets the same value. In this way, both servers will go through the same sequence of state transitions and produce the same output. The master sends its output to the client. The backup verifies it would produce the same output as the master, and then discards its output.

If the master fails to produce output, or if it detects an internal error, the backup takes over. The backup can take over immediately since it is already in the same state as the master was before the master failed. The backup simply stops discarding its output. This is how HotSwap achieves transparent failover: the backup produces exactly the same output as the master would at any moment but discards its output until the master fails.

Both servers must start in the same initial state. To start the system, the two HotSwap processes synchronize their file systems then execute their server programs. After a failure, a new backup server can synchronize files without interrupting the surviving master. The operator can later choose when to restart the master and new backup to achieve full fault tolerance again.

Details of Synchronizing State

Synchronizing server state is critical. HotSwap requires that a server will produce the same output if it receives the same input from a particular set of sources. HotSwap synchronizes system calls like time(), getpid(), socket(), recv(), send(), etc. These are all the inputs used by our pilot servers, and they are enough to ensure that the servers we have tested are

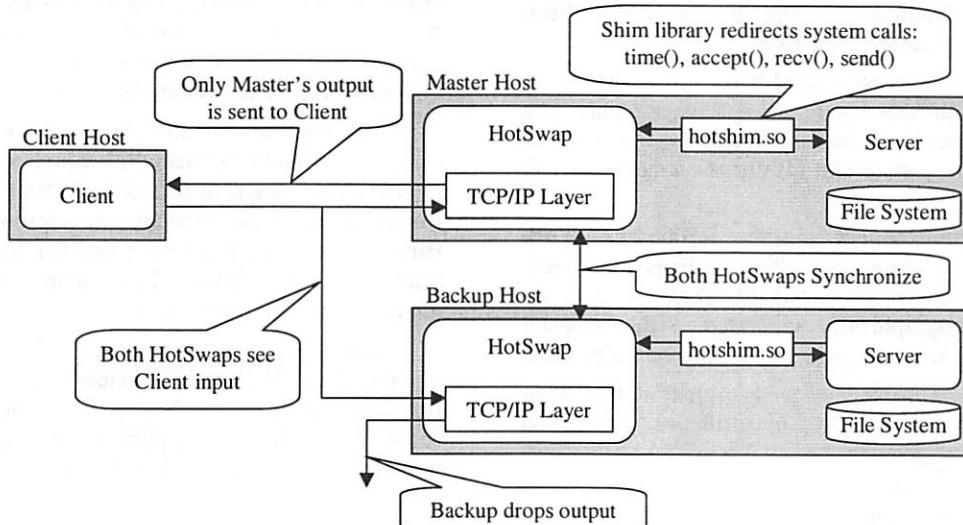


Figure 3: How HotSwap works.

synchronized. Some servers may use other sources of input like inode numbers or direct access to hardware that HotSwap cannot catch or synchronize. HotSwap may not be able to synchronize these servers, but it is likely that the servers themselves could be modified slightly to work with HotSwap to synchronize.

HotSwap synchronizes system calls by relinking programs before they run. In UNIX and Windows, applications are dynamically linked against libraries that provide system calls. HotSwap inserts a shim library that redefines system calls to transfer control to the running HotSwap program. HotSwap also has to synchronize network traffic at the TCP/IP level. Clients and servers usually communicate over TCP, a network protocol that breaks a stream of data into packets that are reassembled in sequence and acknowledged. The operating system (OS) is responsible for managing TCP connections. When a program calls `send()`, the OS adds that to the outgoing TCP buffer, sends a TCP packet and changes the TCP connection state. A TCP connection has many state variables, including packet sequence numbers, timeouts, buffered packets, and acknowledgements. HotSwap must synchronize these state variables, so it uses its own TCP/IP network stack.

HotSwap constantly intercepts and synchronizes a subset of system calls, just enough to ensure synchronization between processes. It also runs in a file-system-only mode where it just synchronizes changes to the local file system. This allows a backup to maintain an active backup of the master's file system without incurring the extra overhead of full process synchronization. This mode is used for disaster with a geographically remote backup.

Limitations

HotSwap relies on each server receiving input only from the set of system calls that HotSwap monitors and synchronizes. HotSwap synchronizes all system calls to sockets, time, file stats, process ids, semaphores, and the `/dev/random` device. HotSwap cannot synchronize state if a server receives input from hardware devices or from the timing of asynchronous signals.

The master and backup systems must start at the same time with identical file systems to ensure they receive the same input from local files. HotSwap runs `chroot()`'ed in a server directory to minimize the amount of files required to synchronize. The `chroot()` has the extra advantage of improving security by limiting the server's access to the file system.

After a server fails, the survivor will continue running by itself as a solo master. A new backup system will continue to synchronize files with the running master, but will not synchronize applications until the running master restarts. If the running master fails, the backup will restart with a synchronized file system, but existing connections and transactions will be lost.

Results

HotSwap has been tested with Perl, Java, Python, Apache, OpenSSL, OpenSSH, and PostgreSQL under Linux.

The first test was replicating a simple web server written in Perl to serve video files. The master was disconnected in the middle of a video, and the client continued displaying the video from the backup without

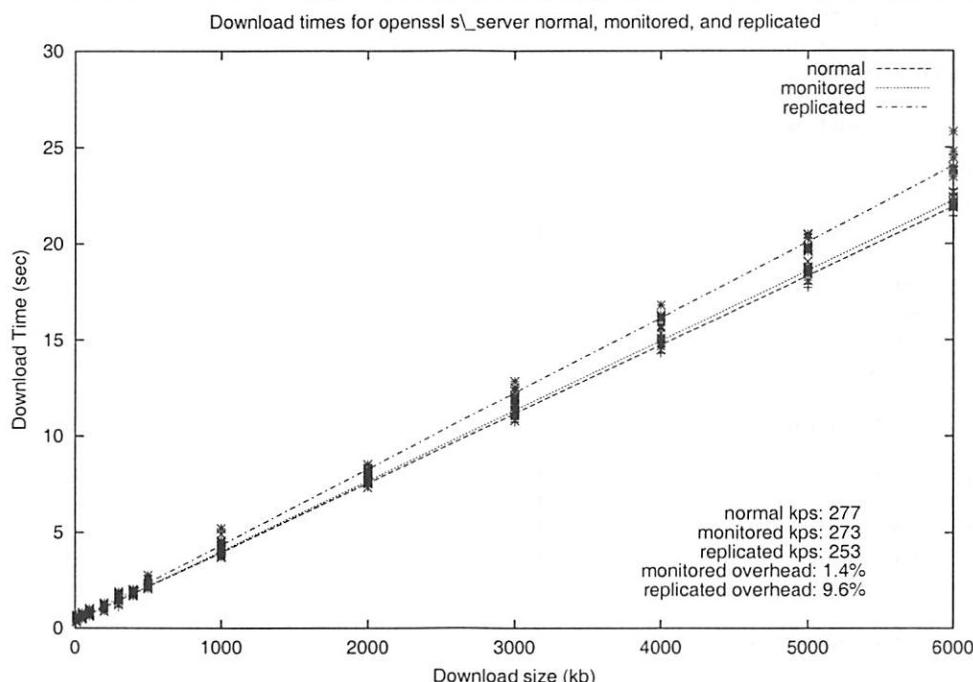


Figure 4: Download performance.

interruption. The same test was then performed using Java, Python, and Apache with good results.

The OpenSSL s_server program was tested to evaluate an encrypted web server. Encrypted connections are usually difficult to replicate, since each side must maintain identical encryption state, and that changes with every byte processed on an encrypted stream. However, as long as the master and backup use the same seed values to generate their initial session keys, they should maintain the same state. Unfortunately, our initial test with OpenSSL failed! Close examination of the OpenSSL source revealed that OpenSSL used an uninitialized buffer plus bytes from /dev/urandom for a seed. This highlights a limitation of HotSwap; processes must use only input from synchronized system calls. Fortunately, we easily patched the OpenSSL server to initialize its buffer and use only /dev/urandom, and the test succeeded. We tested s_server to measure the overhead for just intercepting and monitoring system calls and full replication. We measured the time required to download various sizes of files to see how the overall bandwidth of the server was impacted by replication, using commodity hardware.

The results show that intercepting and monitoring system calls only introduces a 1.4% overhead, and full replication to another box reduced bandwidth by only 9.6%.

The OpenSSH tests demonstrated that HotSwap really does provides a single system image where master and backup appear as one computer. We used ssh to log in and edit files and scp to upload files. All these actions were replicated on both the master and backup simultaneously and transparently.

Replicating PostgreSQL demonstrated that HotSwap can immediately add transparent failover to a database without modifying the database itself.

Conclusion

HotSwap has unique properties. It adds transparent failure and a single system image to servers without any shared components. Backup servers maintain identical file-system and internal memory states with the master. Client connections are never lost or broken on server failure. Servers do not have to be modified, with few exceptions. The price of fully transparent replication is a small amount of overhead.

Availability

HotSwap will be available in server kits that include a complete tested server and the minimal root file system required to support them. We expect HotSwap server kits to be available for download from www.hotswap.net in October, 2002.

About The Author

Noel Burton-Krahn received his M.Sc. in Comp. Sci. from the University of Victoria in 2002, and his

B.Sc. in 1994. He spent the last six years designing and implementing Internet server applications, with emphasis on network protocols, security, and process migration. He has recently founded HotSwap Network Solutions, where he may be reached at noel@hotswap.net.

References

- [Aghdaie01] Aghdaie, Navid and Yuval Tamir, “Client-Transparent Fault-Tolerant Web Service,” *20th IEEE International Performance, Computing, and Communications Conference*, pp. 209-216, <http://citeseer.nj.nec.com/aghdai01clienttransparent.html>, 2001.
- [Alvisi00] Alvisi, L., T. C. Bressoud, A. El-Khashab, K. Marzullo, and D. Zagorodnov, “Wrapping server-Side TCP to Mask Connection Failures,” Technical Report, Department of Computer Sciences, The University of Texas at Austin, <http://citeseer.nj.nec.com/alvisi01wrapping.html>, July 2000.
- [Coulouris01] Coulouris, George, Jean Dollimore and Tim Kindberg; *Distributed Systems: Concepts and Design, Third Ed.*, Addison-Wesley, ISBN 0201-619-180, <http://www.cdk3.net/>.
- [Daniel99] Daniel, Eric and Gwan S. Choi, “TMR for Off-the-Shelf Unix Systems,” *The 29th International Symposium on Fault-Tolerant Computing*, Madison, Wisconsin, USA, June 15-18, <http://www.crhc.uiuc.edu/FTCS-29/pdfs/daniele2.pdf>, 1999.
- [Litzkow97] Litzkow, Michael, Todd Tannenbaum, Jim Basney, and Miron Livny, “Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System,” University of Wisconsin-Madison Computer Sciences Technical Report #1346, <http://www.cs.wisc.edu/condor/doc/ckpt97.ps>, April 1997.
- [Patiño00] Patiño-Marténez, M., R. Jiménez-Peris, B. Kemme, and G. Alonso, “Scalable Replication in Database Cluster,” *14th International Symposium on Distributed Computing (DISC)*, Toledo, Spain, <http://www.inf.ethz.ch/department/IS/iks/publications/pjka00.html>, October 2000.
- [Skoglund00] Skoglund, E., C. Ceelen, and J. Liedtke, “Transparent Orthogonal Checkpointing Through User-Level Pagers,” *Ninth International Workshop on Persistent Object Systems (POS9)*, Lillehammer, Norway, <http://www.l4ka.org/publications/files/l4-checkpointing.pdf>, September 2000.
- [Snoeren01] Snoeren, Alex C., David G. Andersen, and Hari Balakrishnan, “Fine-Grained Failover Using Connection Migration,” *Proc. of Third USENIX Symposium on Internet Technologies and Systems (USITS)*, <http://citeseer.nj.nec.com/snoeren01finegrained.html>, 2001.
- [Wiesmann00] Wiesmann, M., F. Pedone, A. Schiper, B. Kemme, and G. Alonso, “Understanding Replication in Databases and Distributed Systems,” *20th International Conference on Distributed Computing*

- Systems (ICDCS)*, Taipei, Taiwan, Republic of China, <http://www.inf.ethz.ch/department/IS/iks/publications/wpska00.html>, April 2000.
- [Zandy99] Zandy, Victor C., Barton P. Miller, and Miron Livny, “Process Hijacking,” *The Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC’99)*, Redondo Beach, California, pp. 177-184, <http://www.cs.wisc.edu/paradyn/papers/#hijack>, August 1999.
- [Zandy01] Zandy V. and B. P. Miller, “Reliable Sockets,” Computer Sciences Technical Report, University of Wisconsin, ftp://grilled.cs.wisc.edu/technical_papers/rocks.pdf, June 2001.

Over-Zealous Security Administrators Are Breaking the Internet

*Richard van den Berg – Trust Factory b.v.
Phil Dibowitz – University of Southern California*

ABSTRACT

As the security threats on the Internet are becoming more prevalent, firewalls and other forms of protection are becoming more commonplace. Unfortunately, improperly configured firewalls can cause a variety of problems. One particularly nasty problem is when a firewall administrator chooses to use – or continue using – Path MTU Discovery (a good choice in most situations), but blocks packets required for the protocol to work: ICMP type 3 code 4 packets. This problem, the Path MTU Discovery Black Hole, has been discussed many times before. However with under- 1500 MTU protocols such as PPPoE becoming common for both home and business high-speed connections, this problem is affecting more people than ever before.

Introduction

With the rise of security threats due to hackers, script kiddies, and viruses, the use of firewalls is becoming more widespread. This is a positive trend, but as adding firewalls to a network becomes a more common task, other problems inevitably arise. Configuring firewalls without proper knowledge of networking protocols can keep out more than one bargained for. This paper describes one common problem caused by applying overly strict packet filters incorrectly. Causes are examined and solutions are presented and analyzed.

To Filter or Not To Filter

Firewalls in their most simple form are IP routers that can be told which packets to forward and which packets to drop. This task is generally called packet filtering. Deciding what to filter and what not to filter is the hardest part of setting up a firewall. Unless the exact makeup of a network and all its IP applications is known, trial and error is the only way to find out which traffic should be allowed through. Since the idea is to increase security, denying everything unless specifically allowed is the general policy. Having a detailed knowledge of the network you are attempting to protect is critical to deploying an effective firewall.

Internet Control Message Protocol

Certain applications have proven to be more dangerous than others. In the early 90's, most vulnerabilities were found in programs like sendmail and ftpd. Filtering access to these programs was not always possible since they provided a direct service to end users. Instead, an upgrade of the software was needed to divert the attention of crackers elsewhere. In 1996 after the release of Windows 95 another type of problem surfaced. The ping of death revealed an oversight of many

operating system vendors to check the validity of an Internet Control Message Protocol (ICMP) echo request packet. This caused many machines to crash. Later in 1998, the smurf attack used ICMP echo requests to flood a network by pinging a broadcast address. Since ICMP does not directly offer a service, filtering out ICMP packets seemed like a reasonable option to prevent these attacks. This completely ignored the function of ICMP in the TCP/IP suite. The main purpose of ICMP packets is error handling: letting a host know when there is a problem in the communication. The ICMP echo (ping) function can be used for debugging but is in fact far less critical.

Path MTU Discovery Black Hole

When two hosts set up a connection over the Internet using the TCP protocol, each end may let the other know what its maximum segment size (MSS) is. This MSS is derived from the maximum transfer unit (MTU) of the local interface by subtracting 40 bytes for the TCP/IP header. If somewhere along the way an IP packet does not fit in the MTU of the next link, the router handling the packet will fragment it. That is, if Path MTU Discovery is not used.

Fragmenting packets puts a strain on Internet routers, and it also degrades the overall performance of a connection. To overcome these problems, Path MTU Discovery (PMTUD) was proposed in 1988. It is now an Internet standard described in RFC 1191 [1]. PMTUD states that when two hosts communicate over TCP, the Don't Fragment (DF) bit is set. This forces a router that wants to send a large packet over a link that is too small to drop the packet and notify the sending host by sending an ICMP type 3 code 4 message. This message says the destination is unreachable, because your packet is too large and I may not fragment it. In

addition to this standard ICMP message RCF 1191 adds to it: the MTU of the next link is x bytes. This way the sending host can adjust the MSS for the connection and re-send the data.

Since 1988 almost all operating systems have adopted the recommendations of RFC 1191 and use Path MTU Discovery when communicating via TCP. A problem arises when PMTUD is enabled, but incoming ICMP type 3 code 4 messages are filtered by a firewall. Since the sending host is never properly notified of any problem with the size of the packets, it will not adjust its MSS. Communication with the other host will fail. This is known as the Path MTU Discovery Black Hole problem and is described in detail in RFC 2923 [2].

The problems with PMTUD and ICMP filtering date from long before RFC 2923. One example is Path MTU Discovery and Filtering ICMP [10] explaining the issue as early as January 1998. On mailing lists like the North American Network Operators' Group (NANOG) the problem has been discussed extensively, and questions about it return every few months.

This is because more and more people are being affected by the black hole.

Home and Business Networks and the Black Hole

Links with small MTU sizes are quite rare in core of the Internet. This is perhaps why filtering out all ICMP packets does not seem to cause immediate problems. However, it is causing problems with networks behind newer broadband connections in both homes and businesses (older technologies such as SLIP and X.25 are also vulnerable). Techniques like xDSL and DOCSIS (data over cable TV service) provide an Internet connection that is always on. Combined with the large bandwidth offered by these services, connecting multiple computers to the uplink becomes feasible and rewarding. The high bandwidth also requires the need of connecting over a faster medium than a serial cable (being either RS-232 or USB). Often Ethernet is chosen with PPP over Ethernet (PPPoE) as the WAN protocol. PPPoE uses encapsulation to deliver IP packets destined for the Internet to the broadband modem via Ethernet.

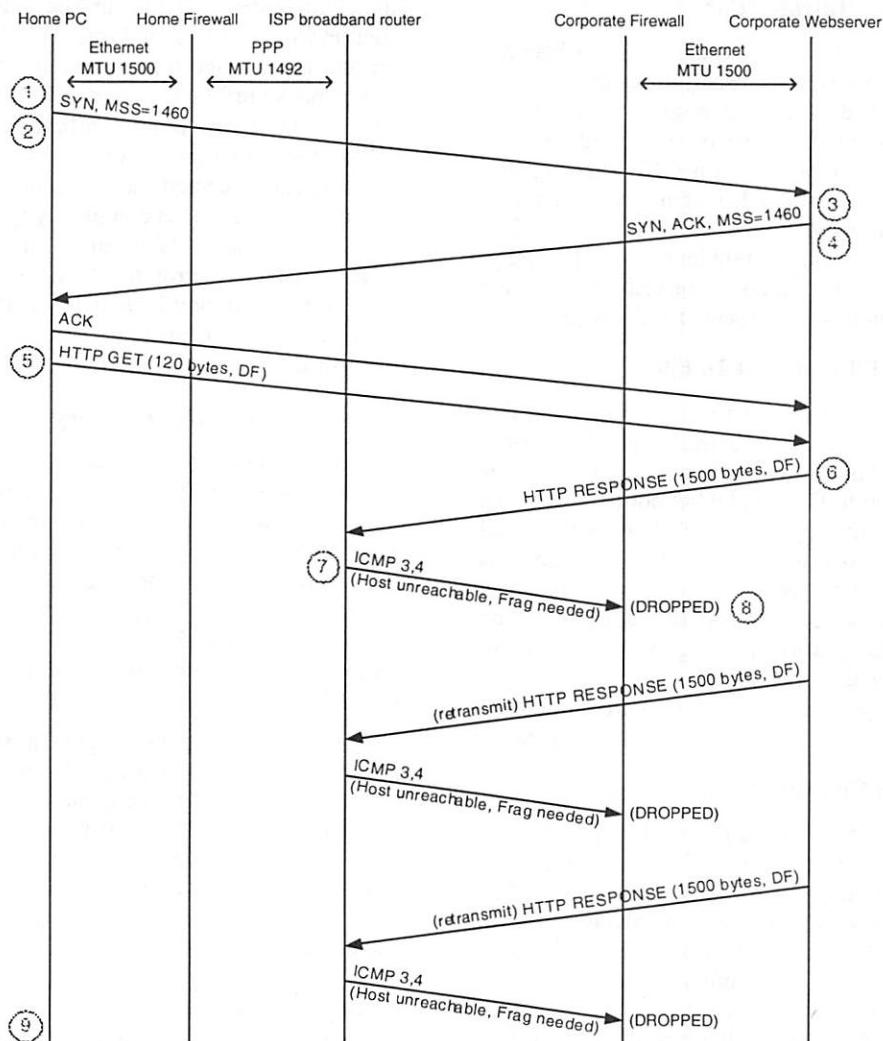


Figure 1: IP connection affected by the Path MTU Discovery Black Hole.

Going back to our Path MTU Discovery Black Hole problem, the MTU of the PPP interface will have to allow for the encapsulation so that the total PPPoE packet will fit in the standard Ethernet MTU of 1500. PPPoE interfaces therefore have a standard MTU of 1492. The disaster scenario now becomes clear:

1. A workstation on the network will start a TCP session to, say, a web server on the Internet
2. The PC sets the MSS to 1460 since the Ethernet MTU is 1500
3. The web server also connects to Ethernet, so it replies with an MSS of 1460
4. The web server enables PMTUD for the traffic to the PC
5. The PC sends an HTTP request (typically a few hundred bytes)
6. The web server starts sending the requested file, in 1500 bytes IP packets
7. The broadband router at the ISP of the end network cannot fit the packet into the PPP link and sends an ICMP type 3 code 4 message to the web server
8. A firewall between the end network and the web server drops the ICMP packet (often this is the firewall meant to protect the web server, but it can easily be any other firewall or router in between the two end networks)
9. The user is unable to browse the web site

The example uses web browsing and HTTP, but it holds true for any TCP communication sending messages of more than 1452 bytes at a time (E-mail, ftp, etc.).

Figure 1 shows a home firewall. This can either be a device specially designed for this purpose, or a generic workstation configured for this task – and could just as well be the firewall of a business. With the always-on feature of broadband connectivity, setting up a firewall at home is becoming a must.

Who Is (Not) Affected

A link with a small MTU can exist anywhere on the Internet. So theoretically everyone can be affected by this problem. As explained earlier, home and business networks utilizing PPPoE or similar protocols common in today's DSL and cable networks have a higher chance of encountering this problem. This increased probability does not apply to the following setups:

- **Just one workstation connected to a modem.**

Since the MTU of the PPP interface is used to calculate the MSS for each TCP connection, the web server will only send packets that will fit into the PPP link.

- **Home gateways with a public IP address on an Ethernet interface.** The external Ethernet interface can directly be used for Internet traffic since it has its own public IP address. Since no encapsulation is needed, the MTU used for Internet traffic is 1500 (the default Ethernet MTU).

- **Home gateways connecting to a modem using USB.** Since USB does not have an MTU of its own, the PPP connection can safely use an MTU of 1500 or higher.

- **Home gateways connecting to a modem using PPTP.** The Microsoft Point-to-point Tunneling Protocol (PPTP) uses a modified version of Generic Routing Encapsulation (GRE). The MTU of the GRE interface is set to 1500. Since GRE adds 56 bytes of overhead to each packet, it is possible packets will not fit into the MTU of Ethernet. In such case, the original IP packet is fragmented even if the Don't Fragment bit is set. This is quite nasty and lowers performance [1]. It does however prevent the Path MTU Discovery Black Hole from occurring on account of the PPTP link.

Cause of the problem

As mentioned above, the Path MTU Discovery Black Hole problem is caused by using PMTUD without allowing crucial ICMP packets to pass network filters. RFC 2923 [2] describes this as an act of overzealous security administrators. It is a sign of the times to have very strict firewall policies. Check Point Software Technologies is the undisputed market leader in firewall solutions. Their FireWall-1 product used to ship with the default Policy Properties containing a setting to allow all ICMP traffic to pass. When the smurf attack hit in 1998, Check Point was publicly criticized for allowing ICMP through by default. This caused the company to change the default settings to disallow all ICMP traffic. Since Path MTU Discovery has become a standard TCP/IP feature, when anyone now installs an out-of-the box Check Point firewall, they introduce the PMTUD Black Hole. It is now left to the security administrator to explicitly allow ICMP type 3 code 4 packets to the servers that use Path MTU Discovery, or turn off PMTUD if they are uncomfortable with allowing such ICMP packets into their network.

RFC 2923 [2] mentions in Chapter 3:

It is vitally important that those who design and deploy security systems understand the impact of strict filtering on upper-layer protocols. The safest web site in the world is worthless if most TCP implementations cannot transfer data from it.

We could not have said it any better. Many authoritative sources confirm that allowing ICMP type 3 code 4 packets through a firewall does not pose a security risk [3, 9].

Size of the Problem

If all of the above still sounds like a mere academic problem, here's the scary part: not only less experienced administrators are blocking all ICMP packets while using Path MTU Discovery. Web sites of organizations with a focus on security also have this problem. Just to name a few:

- www.securityfocus.com (recently fixed)
- www.cert.org
- www.verisign.com
- www.counterpane.com
- www.ntsecurity.com

If you cannot trust such security experts to correctly configure a firewall, whom can you trust? Is it fair to refer to this behavior as less experienced? Could not these administrators be ahead of the game by filtering something that may soon become a security hole? In fact, there is a way for administrators to successfully block ICMP type 3 code 4 packets from entering their network without breaking things. Blocking these packets without taking the proper precautions however, is not acceptable for security professionals administering firewalls. Proper solutions are discussed below.

Solutions

Since this problem has been around for quite a while, different solutions have been developed. Interestingly enough, even though the problem is caused by misconfigurations at the server side, most solutions are aimed at modifications of the clients. Apart from the moral discussion of this, it makes little sense implementation wise. If one popular server is misconfigured, all users behind a small MTU link wishing to use this server will have to adjust their settings. It would be much easier if the users could convince the maintainers of the broken site to solve the problem at its source. The truth is that this is not an easy task. This is why solving the issue at the client side is so popular.

The first three solutions we present depend on the cooperation of the (security) administrators of the websites with a misconfigured firewall. Only if this cannot be achieved, should one look at the things that can be done on the client side.

Allow ICMP Type 3 Code 4 Packets To Reach the Servers

The simplest solution is to allow Path MTU Discovery to work as it was intended: set the Don't Fragment bit on all packets and allow ICMP type 3 code 4 messages to reach the server. This means changing the overly strict rules on firewalls and other active packet filters. It should be noted that this is not considered a security risk by many authorities [3]. However, if a firewall administrator feels that allowing such packets is more risk than it is worth, there are other solutions.

Disable Path MTU Discovery

If allowing ICMP into a network is not an option or cannot be achieved, the right thing to do is disable Path MTU Discovery on all servers that cannot receive ICMP type 3 code 4 packets. Since receiving these packets is a requirement for PMTUD to work [1], it breaks RFC standards and simply makes no sense to have PMTUD enabled on these servers. How to disable this feature depends on the operating system

of the server. Cisco published a page with setting for some popular operating systems [4]. It is worth noting that disabling PMTUD to solve the PMTUD Black Hole will cause fragmentation. PMTUD was introduced to maximize performance by minimizing fragmentation [1]. Reintroducing fragmentation should be considered only if the previous solution is not feasible.

Path MTU Discovery Black Hole Detection

§2.1 of RFC 2923 [2] recommends the implementation of a PMTUD Black Hole detection mechanism. This is done by turning off the DF bit when retransmitting TCP packets. Various TCP/IP stacks now implement this detection scheme, but it is not turned on by default. The very nature of this solution (retransmissions) results in lower performance. Since it requires changes on the server side anyway, it makes more sense to turn off Path MTU Discovery altogether.

Using a Proxy Server

If a server is suffering from the Path MTU Discovery Black Hole, and it cannot be fixed there are some things that can be done on the client side that will prevent the Black Hole from acting up. For web browsing for example, it is possible to use a proxy server that does not suffer from the PMTUD problem. The proxy will then retrieve the pages on the client's behalf, repacking it into smaller TCP packets. Of course this only solves the problem for protocols that can be proxied.

Lowering MTU/MSS of the Internal Network

Another option is to lower the MTU of the client to the MTU of the smallest link between the client and the server. This way, the client will advertise a smaller MSS indicating to the server that its packets should not exceed this size. The same result can be achieved by lowering the maximum MSS value that a host will advertise [4]. This solution will not solve all problems. While, the MTU of the uplink is probably known and can be used as a guideline for the MTU of the systems on the LAN, one cannot be sure that this will always be the smallest MTU of the path between the clients and a server. If a smaller MTU exists on this path, ICMP type 3 code 4 messages will be sent to the server and the connection will still fail. Additionally, non-TCP protocols like UDP and IPSec will still suffer from the PMTUD Black Hole.

MSS Clamping

The solution of lowering the MTU on all systems of the LAN sounds feasible when all means less than five. If there are a dozen or more systems, this becomes a rather gruesome task. Several solutions exist to automatically adjust the MSS of TCP packets when they are being routed by the internal gateway. This is a particularly nasty solution. Per definition a router should not interfere with end-to-end settings like the MSS. Additionally, some protocols like IPSec will break when the MSS is changed in midcourse.

- There are several implementations of this hack:
- --clamp-mss-to-pmtu switch for IPTables in Linux 2.4.x kernels [5]
 - CLAMPMSS setting of Roaring Penguin's PPPoE Software [6, 13]
 - mssfixup command of ppp for FreeBSD [7]

This solution suffers from the same problems as above: there is no guarantee that the uplink MTU is the smallest in the path (even if it is, this only works for TCP).

The MSS Initiative

In an attempt shift the focus to the cause of this issue rather than the effect, we started The MSS Initiative [8]. The purpose of this initiative is to raise awareness of systems administrators about the Path MTU Discovery Black Hole problem. We believe that when enough security administrators realize that blocking ICMP type 3 code 4 packets breaks one of the core IP protocols, they will adjust the rule sets of the devices they manage or turn off PMTUD. Gruesome hacks like MSS clamping will then become unnecessary. The MSS Initiative maintains a list of sites that are currently suffering from the Path MTU Discovery Black Hole and attempts to notify the administrators of those sites. This works in two ways: end users can check the list to see if a site they cannot reach is misconfigured, and hopefully administrators will take action upon receipt of the notice they receive from us. We also offer to help administrators unsure of how to fix their setup.

Determining if a site suffers from the Path MTU Discovery Black Hole can be difficult. It is very easy to mistake other network problems for this one. Users are encouraged to follow the instructions detailed on The MSS Initiative website if they believe a site is suffering from the PMTUD Black Hole. Users may then report the site to the Initiative so it can be added to the list and the administrator contacted.

Conclusion

Packet filters and firewalls have become necessary tools to protect systems against the growing hostility on the Internet. At the same time these tools themselves, if not configured properly, pose as a threat against one of the core protocols of the IP suite. In an ideal world, everyone would follow the guidelines set forth by Internet standards and RFCs. In a diverse and disjoint society like the Internet this cannot be expected to happen. However, when some of these standards are violated by a large number of sites and even some important vendors and security specialists fail to follow them correctly, things do break. It is in the nature of the users of the Internet to find a way around the problems that arise. Fixing things locally is attractive because of the speed and control that can be achieved, but it also allows the real problems to persist.

It is time make an effort to correct the problem that has been explained in this paper. If we do not, we might have to abandon the usage of Path MTU Discovery in the near future. This is neither efficient nor practical since it is also one of the core protocols of IPv6 [11, 12].

About the Authors

Phil Dibowitz is a junior at the University of Southern California studying Computer Engineering and Computer Science. He has held positions as a Solaris, Linux, and Netware Systems Administrator, and as a Network Administrator. His main interests are network security, networking, and UNIX. Phil maintains the official IP Filter FAQ and is the author of open source software called IPTState (monitoring software for IP Tables). Resume, projects, and accomplishments can be found at <http://home.earthlink.net/~jaymzh666/>. Phil can be reached at phil@ipom.com .

Richard van den Berg has been working as a networking consultant for many large telcos and ISPs in The Netherlands since 1997. He was one of the networking specialists at Sun Microsystems' Professional Services division and currently works for Trust Factory as an independent Security Architect. Richard has the most fun decoding TCP/IP packets at the bit level while writing networking tools in various programming and scripting languages. Richard can be reached at richard@trust-factory.com .

Together the authors founded the MSS Initiative [8].

Acknowledgements

We would like to thank Chris Goggans, Marcus Ranum and the anonymous reviewers for their early reading and useful comments. We would also like to thank Joep Vesseur of Sun Microsystems for suggesting the diagram used in Figure 1. Richard likes to thank his wife Jaya Baloo for her love and support.

References

- [1] Mogul, J., S. Deering, *RFC 1191 Path MTU Discovery*, <http://www.ietf.org/rfc/rfc1191.txt>, November 1990.
- [2] Lahey, K., *RFC 2923 TCP Problems with Path MTU Discovery*, <http://www.ietf.org/rfc/rfc2923.txt>, September 2000.
- [3] van Eden, L., *The Truth About ICMP*, <http://rr.sans.org/threats/ICMP.php>, May 2001.
- [4] Cisco Tech Notes, *Adjusting IP MTU, TCP MSS, and PMTUD on Windows, HP and Sun Systems*, <http://www.cisco.com/warp/public/105/38.shtml>.
- [5] Hubert, B., G. Maxwell, R. van Mook, M. van Oosterhout, P. B. Schroeder, J. Spaans, *Linux Advanced Routing & Traffic Control HOWTO*, <http://www.linuxdoc.org/HOWTO/Adv-Routing-HOWTO.html>, December 2001.

- [6] *Roaring Penguin's PPPoE Software*, <http://www.roaringpenguin.com/pppoe/>.
- [7] *FreeBSD System Manager's Manual for ppp(8)*,
<http://www.freebsd.org/cgi/man.cgi?query=ppp&sektion=8>.
- [8] Dibowitz, P., R. van den Berg, The MSS Initiative, <http://home.earthlink.net/~jaymzh666/mss/>, February 2002.
- [9] Arkin, O., *ICMP Usage In Scanning*, Black Hat Briefings 2000, Amsterdam, <http://www.sys-security.com/html/papers.html>.
- [10] Slemko, M., *Path MTU Discovery and Filtering ICMP*, <http://www.worldgate.com/~marcs/mtu/>, January 1998.
- [11] Deering, S. and R. Hinden, *RFC 2460 Internet Protocol, Version 6 (IPv6) Specification*, <http://www.ietf.org/rfc/rfc2460.txt>, December 1998.
- [12] McCann, J. and S. Deering, *RFC 1981 Path MTU Discovery for IP version 6*, August 1996, <http://www.ietf.org/rfc/rfc1981.txt>.
- [13] Skoll, D., "A PPPoE Implementation for Linux," *Proceedings of the Fourth Annual Linux Showcase & Conference*, Atlanta, <http://www.usenix.org/publications/library/proceedings/als2000/skoll.html>, October 2000.

An Approach for Secure Software Installation[†]

V. N. Venkatakrishnan, R. Sekar, T. Kamat, S. Tsipa and Z. Liang
– Computer Science Dept., SUNY at Stony Brook

ABSTRACT

We present an approach that addresses the problem of securing software configurations from the security-relevant actions of poorly built/faulty installation packages. Our approach is based on a policy-based control of the package manager's actions and is customizable for site-specific policies. We discuss an implementation of this approach in the context of the Linux operating system for the Red Hat Package manager (RPM).

Introduction

Management of software installations has been one of the biggest problems facing system administrators.¹ Significant progress [6] has been made in some aspects of this problem, e.g., management of dependencies and conflicts among packages. In other areas that concern overall system security and interoperability with existing applications, package managers still fall short, as they make several unrealistic assumptions:

- *System administrators want to treat packages as “black boxes.* Although system administrators are not interested in the details of installation, they certainly care about package installation actions that have implications for overall system security and operation, e.g., addition of new users, modifications to boot-time scripts, addition of entries to crontab, modifications of system libraries, changes to global configuration files (e.g., /etc/inetd.conf) or application-specific configuration files or in the case of Windows, changes to system registry.
- *Package installation steps operate correctly.* Few provisions exist for dealing with poorly-written packages that crash in the middle of the installation process. Typically, such crashes would result from unanticipated conditions (involving the configuration of the system on which the package is being installed) encountered by install scripts that are included with many packages. System administrators are all too familiar with situations when packages can neither be fully installed, nor be fully uninstalled, leaving the system in an inconsistent state.
- *All software and system configuration updates are the result of package installation:* In practice, however, system administrators have to frequently

configure systems or packages by manually editing config files. In addition, software is frequently installed outside of the package management system, e.g., by downloading and compiling a source-code archive. Package managers interact poorly with such situations, often ending up overwriting critical application files. Although package managers can save backup copies of certain config files, the sysadmin is usually not alerted that the config files have been updated.

We describe a new approach that augments existing package managers such as RPM to overcome the above problems. Our approach enables system administrators to reason about the security-critical actions of an installation/upgrade process, check whether these actions are compatible with specified security policies, and if so, allow the installation to proceed. During the installation, all actions are logged so that they can be rolled back in the event of an installation failure.² We have implemented our approach in the form of a tool called *RPMShield* that operates in conjunction with RedHat's package manager (RPM). As compared to existing package managers, our approach offers the following benefits:

- *Policy-based control of package installation actions.* While existing package managers such as RPM allow a system administrator to examine package contents and installation scripts in detail, this is a cumbersome process and hence seldom undertaken. In contrast, our approach presents a convenient interface through which a system administrator can exert control over installation actions that impact system security or the operation of existing applications.
- *Interoperability with changes made outside of package managers.* Our approach provides a

[†]This research is supported in part by a ONR University Research Initiative grant N000140110967 and NSF grant CCR-0098154.

¹In this paper, we use the term “system administrator” to refer to professionals and end-users that perform software installation.

²Note that a complete rollback is impossible if the installation scripts communicate over the network, or when processes unrelated to the installation are allowed to make system changes after reading files modified by the installation process.

convenient mechanism to control updates to manually edited files, files shared among multiple packages, or more generally, files installed outside the scope of the package manager.

- *Normal-user installation of packages.* Individual users often want to install packages that are of interest to themselves. Since all RPM installation actions require super-user privilege, normal users are unable to install such packages for themselves. Our approach can support this capability through the use of security policies that limit installation actions so that the changes are restricted to a specific user directory.
- *Tolerance to failures.* Package managers offer poor support to revert to original system configuration when an installation upgrade/process fails. Our automatic recovery mechanism reverts the system to its original (consistent) state.

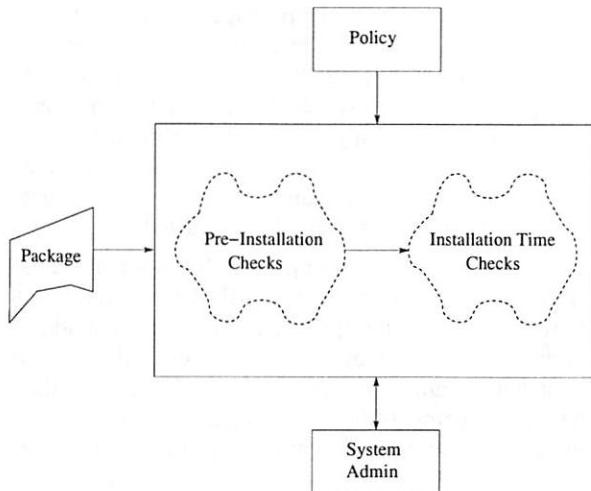


Figure 1: Approach overview.

We note that this paper is concerned with the package installation process, and does not address security implications of running the applications installed as a result. Ensuring safety of the system while supporting the execution of untrusted application is an orthogonal problem. This is the subject of many papers on digital

signatures [7], sandboxing [9], proof-carrying code [11] and model-carrying code [12].

Overview of Approach

Our approach, presented in Figure 1 divides package installation into two phases. 1) In the *pre-installation phase*, a package is analyzed to determine its compatibility with a system administrator's policies. 2) In the *installation phase*, where the actual package installation takes place in a controlled environment. Each of these phases is described below.

Pre-installation Phase

The pre-installation phase (see Figure 2) consists of the following steps:

- *Generation of behavioral models.* In this phase, a package is analyzed to identify the security-relevant actions that will take place during its installation. The analysis involves two steps: 1) finding the list of files the package will install/upgrade, and 2) capturing the intended behavior of its (pre-install and post-install) scripts. The first step involves querying the package itself as well as the packages database. The second step involves learning the behavior of the scripts/make files. More details on the model generation process is presented in the section 'Model Generation'
- *Consistency resolution.* The model generated in the previous step is supplied to the *consistency resolver* which checks whether this behavior is in accordance with the security policy provided by the system administrator. A discussion of security policies that could be enforced though the system is presented in the 'Security Policies' section. Consistency resolution is described in its own section.

By performing consistency resolution before installation, our approach avoids the time-consuming step of actual installation when there is a conflict. In addition, all conflicts with the policy are identified and presented together, which enables a system administrator to make more informed decisions. This contrasts with conflict identification during actual installation, when each conflict must be presented individually to the system

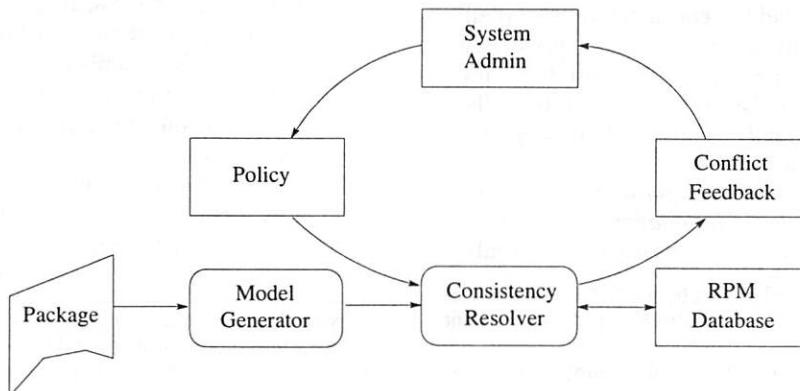


Figure 2: Pre-installation phase.

administrator for acceptance. This cumbersome process can lead to “click fatigue,” sometimes causing the system administrator to make inappropriate decisions.

Installation Phase

While the benefits of pre-installation checks were identified above, it is not always possible to identify all conflicts statically. Scripts may perform complex computations (e.g., creating a file name from several command-line arguments or environment variables) whose results cannot always be statically determined. Such conflicts are dealt with during the second phase of package installation (refer to Figure 2), namely, the installation phase.

In this phase, as shown in Figure 3, the package manager is allowed to run in an environment where the system calls made by the package manager process and its children are monitored by RPMShield. These system calls are compared with the policy provided by the system administrator during the pre-installation phase. Typically there are no policy violations in this phase, as they would have been handled in the previous phase. However, if conflicts do arise, this information is presented to the system administrator. If the violation is accepted, then package installation proceeds. If not, installation is aborted, and the system state is restored as it was prior to the installation. The runtime-checking mechanism is described in the ‘Runtime Interception’ section.

Description

This section elaborates on the various components that were introduced in the preceding high-level discussion.

Security Policies

We use an expressive policy language that, in addition to capturing conventional access-control

policies, can also express context sensitive policies such as “this application cannot modify files owned by other applications,” or history sensitive policies such as “the installation process can only delete the files it has created.” Access control policies and context sensitive policies are specified conveniently through a GUI, as shown in Figure 3. In this figure, the system administrator can specify whether the package installation process can possess the corresponding *capabilities*. The first row describes a capability where a package can create files in directories not owned by itself; In the second row, policy specification for writes to files that are owned by the package, but modified from the original installation (e.g., config files) are shown; and writes to files owned by other packages are shown in the succeeding row.

Similar capabilities that could be specified using the GUI include the ability to add users, perform network operations, update shared libraries, modify system services (e.g., files in the /etc/rc.d/* directories), execution of arbitrary system commands and so on. History sensitive policies are currently not expressed through the GUI, but could be specified using our underlying expressive policy language [13].

These security policies are internally represented as *extended finite state machines*. An finite state machine consists of *states* and *transitions*. The states of these machines correspond to various program points and the transitions are over an alphabet of system calls with their arguments. There are *condition guards* associated with transitions. Whenever, a condition guard is enabled, an optional *action* associated with the guard is triggered. A simple example of an extended finite state automaton is presented in Figure 5, which illustrates the use of these automata in keeping track of the number of bytes written to a file (denoted by MY_FILE).

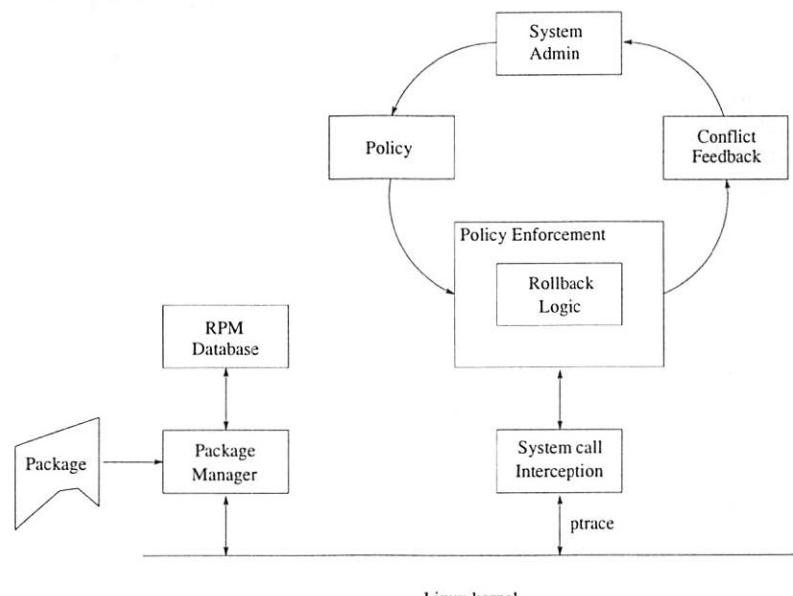


Figure 3: Installation phase.

In this figure, whenever an open system call is made, the condition guard associated with this transition checks whether the file opened is MY_FILE, and if so, the file descriptor is stored in the variable X. Later in the program, if a write operation is performed, the condition guard associated with this transition checks whether the file descriptor equals the descriptor stored in X, and if so, increments the variable count. (For simplicity, we do not show the states and transitions corresponding to the invocations of the close system call). The policy represented thus is a simple example of a history sensitive policy, and the variables that are associated with the transitions enable such policy specifications.

For more information on our work in compiling high level specifications into extended finite-state machines, we refer the reader to [13].

Model Generation

Model generation involves determining the security relevant actions of the scripts and obtaining the list of files the package plans to modify/upgrade. The latter is obtained directly by querying the package, so we describe the analysis of scripts in the following section.

Scripts

There are several approaches that address the problem of analyzing a shell script to determine its behavior. A *static analysis based* approach is one which would analyze the actions of a script without executing it. It usually involves parsing the input script, and interpret the script's actions to analyze its behavior. Such an approach could build on modifying the shell interpreter and performing operations in an

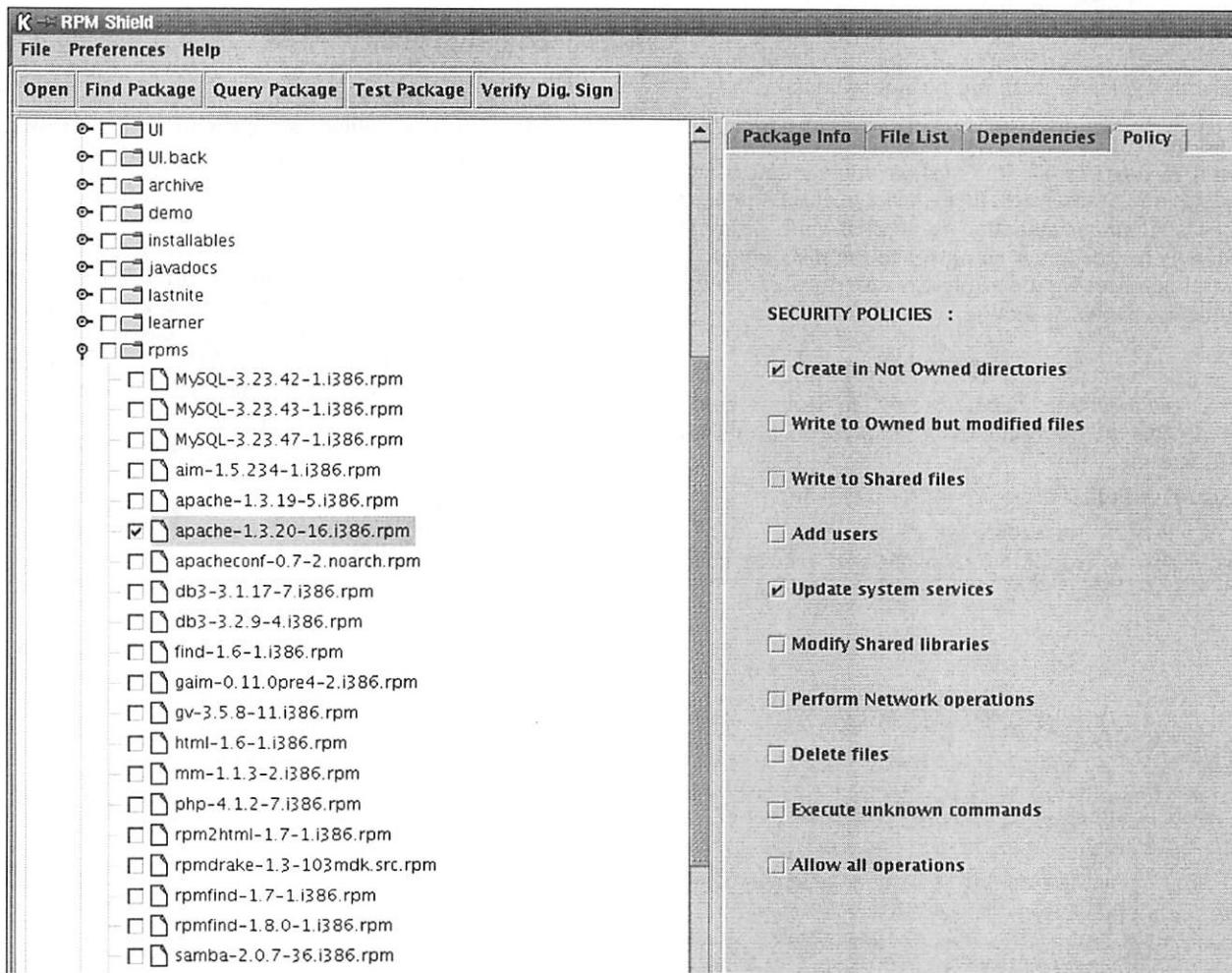


Figure 4: Screen shot of RPMShield.

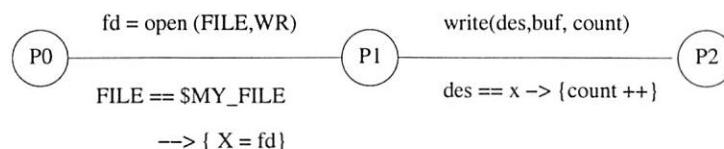


Figure 5: An extended finite state automaton.

abstract domain (a general technique called *abstract interpretation* [8]).

The main advantage of a static analysis based approach is that it has the ability to reason about the actions of the program at a level closer to the program source. However, there are a few disadvantages with such an approach. Shell scripts heavily depend on the environment and redirection for their successful completion. Any static analysis based approach has to *approximate* the environment and the effects of redirection. Thus, the behavior obtained through analysis is usually incomplete.

In addition, there is one more practical implementation problem: The number of shell interpreters that are available in a general purpose system abound. An analyzer that has to deal with an arbitrary script needs to have a front end that would support the idiosyncrasies of the syntax of a number of languages (bash, csh, perl to name a few!). Clearly, this is not a desirable situation.

We follow an alternate approach that follows the *program behavior learning* approach. In this approach, we intercept the system calls of the shell script. We inspect the system call and its arguments, and allow it based on whether it is trying to perform a security-critical operation. We allow access to all operations that are not security-critical: for example, reads to non-sensitive files, creation of temporary files, execution of commands that do not alter the system state and so on. When the program performs write-operations or security critical operations like adding a system service or a user, we simply *fake* the return value of the system call.

By *faking*, we mean returning success without executing the corresponding system call. Thus, the original operation is not performed, and the trace that is generated is the model of the program capturing the intended behavior of the program. All the environment related information is available to this approach (unlike the previous approach). See the ‘Examples’ section for an example of a model that is generated.

Consistency Resolution

The consistency resolution step involves checking of the model that is generated from the previous step against the security policies of interest. This step involves two operations. The first operation involves checking whether the scripts in the package conform to the security policy. The second operation involves checking whether the file creation / update operations of the package is in accordance to the policy.

Script Checking

Checking whether the execution of a script will violate the given security policy is an interesting problem. The model generated for the shell script contains the trace of the script execution. The security policy (discussed in its own section earlier), is represented in the form of an extended finite state automaton. The policy proscribes all the invalid traces. The model,

obtained as an output of the previous step, is presented as an input string to this automaton. If the trace execution is a valid string that is accepted by this automaton, then we can conclude that the execution of this script will violate the specified policy.

As an example, consider an installation script that adds a new user to the /etc/passwd file, while the policy allows no such addition. This conflict information (including the specific action and/or file that caused the conflict) is presented to the system administrator, who may decide to abort the installation, or refine the policy to eliminate the conflict, e.g., permit addition of user to the password file.

Checking of File Updates

The consistency resolver detects any conflicts between the policy and the package behavior model, e.g., if the policy allows updates only to those files owned by a package, then a conflict will arise if the package updates a file that has been updated manually or by a different package. Similar violations are reported for creating files in directories that are not owned by the package, deletion of files and so on.

Runtime interception

The installation phase is realized by the following components.

- *System call interception environment.* System calls and arguments are forwarded to the *policy enforcement engine* using a ptrace-based system call interception facility that we had developed earlier for Linux [10]. This infrastructure provides the facilities for one program to inspect another (target) program whenever the target program performs system calls; check the arguments to the system call; and, if necessary, fake the return value without executing the system call.
- *Policy enforcement engine.* The policy enforcement engine is implemented as an extended finite state machine, which was discussed. These automata enforce these policies with very low overheads (typically under 2%) [13].

In addition, the policy enforcement engine incorporates *rollback logic*, which keeps track of the files modified by the package manager (since the original installation), as well as the original contents of these files. If the installation is to be aborted, then this information is used to reset the system state to what it was before the package installation began.

Other Features

Our tool has a convenient user interface and incorporates attractive usability features, which we describe below.

Query downloads. Most packages have dependencies, i.e., they can be installed only if certain other packages are present in the system. A novel feature in

RPMShield facilitates easy installation of such packages, by downloading them from RPM mirror servers (such as rpmfind.net). The user can configure this set of servers. When an installation encounters dependency requirements, our implementation searches for the existence of these dependency packages on any of the servers.

If the dependency package is found, it is reported to the user and downloaded at his/her discretion. Of course, the downloaded dependency is subjected to the same security checks. Finally, after a successful download, the entire installation process is resumed. RPMShield also takes care of transitive dependencies, e.g., if a package A is dependent on package B which in turn is dependent on package C and so on, then both, packages B and C (and further dependencies) are downloaded and installed first and finally package A is installed.

The implementation of this feature involves the construction of a directed-graph where the nodes represent packages and the edges represent the dependencies between such packages. The installation then starts by installing from the nodes farthest from the root and then proceeds back to the root.

Normal User Installation. Since RPM installations require root privileges, normal users (who do not have root privileges) do not have an opportunity to install packages that are of interest to themselves. Using a highly constrained security policy, we can provide a confined environment where the package can be installed by the normal user. However, not all packages can be installed by a normal user. The packages have to be re-locatable, and must not update any system owned files. Due to the nature of this constrained policies, it is not possible to change them through the graphical-user interface. (One could however change them by modifying the policies in the underlying language).

```
/sbin/chkconfig --add httpd
# safely add .htm to mime types if it is not already there
[ -f /etc/mime.types ] || exit 0
$TEMPTYPES=''/bin/mktemp /tmp/mimetypes.XXXXXX'
[ -z "$TEMPTYPES" ] && {
    echo "could not make temporary file, htm not added to /etc/mime.types" >&2
    exit 1
}
( grep -v "text/html" /etc/mime.types
types=$(grep "text/html" /etc/mime.types | cut -f2-)
echo -en "text/html==>[ignored: t]<====>[ignored: t]<====>[ignored: t]<=="
for val in $types ; do
    if [ "$val" = "htm" ] ; then
        continue
    fi
    echo -n "$val "
done
echo "htm"
) > $TEMPTYPES
cat $TEMPTYPES > /etc/mime.types && /bin/rm -f $TEMPTYPES
```

Listing 2: Post-installation script of apache server.

Examples

We illustrate our approach by running through the installation of the web-server program Apache through our tool. We illustrate the various stages of the installation process through this example.

Apache is a popular and freely-available Web server. The package consists of a set of installation files as well as pre-install and post-install scripts. This example pertains to the version of 1.3.20-16 of apache server.

The system first queries whether the package satisfies all dependency checks. If there are any dependencies on packages that are not yet installed on the system, then the user is queried for downloading of these packages from the download sites. These checks are run through the same security checks as the package. In the following discussion we assume that all dependencies are satisfied.

The pre-installation script is given in Listing 1. The script creates a user in the system. Obviously, this is a sensitive operation, and the system administrator is alerted of this operation. (We do not show the model for this script, but present the model for the post-install script).

```
# Add the "apache" user
/usr/sbin/useradd -c "Apache" -u 48 \
-s /bin/false -r -d /var/www apache \
2> /dev/null || :
```

Listing 1: Pre-installation script.

Listing 2 is the post-installation script of apache server. The generated model is shown in the Figure 6. In this model the states refer to various program points, and the transitions refer to various system calls with their arguments. Due to constraints on the size of the figure, we omit some details such as system call arguments for a selected set of system calls.

This script performs a update to the system services scripts. Also the script updates the file /etc/mime.types that is shared by other applications. The user is alerted of these operations. The other operations performed by the script such a creating and deleting of temporary and running other utilities such as cut, grep etc., are allowed by the security policy.

In addition, suppose if this installation of apache was actually an upgrade from a previous version, then the system keeps track of all the files that have been modified since the previous installation. This not only includes all configuration files, but other files such as local symbolic links, text files, etc. An attempt to delete/overwrite these files is presented to the system administrator. All through the installation, such files are backed up, such that in case the user decides to abort the installation, the system can be reverted to its original state.

Applicability to Other Systems

Although the approach presented in this paper is described only in the context of the Linux operating system and the RedHat package manager, the overall architecture can be ported to other Unix like environments with relative ease. We discuss about migrating to other package managers below.

We have used the Java environment for the implementation of the graphical user interface. Hence this portion of the implementation is portable. The model generation involves querying the package and hence this has to be customized for the particular package format. The consistency checking involves querying the package database and this requires customization as per the package manager's programming interface for querying and modifying its database. The model generation for scripts and the runtime checking steps involve system call tracing. Although, the ptrace system call is not completely portable across different Unix environments, similar facilities exist for other Unix variants, as evidenced by the implementation of gdb-like debuggers for these systems. In fact, our system call interceptor [10] provides a uniform programming interface implementation that abstracts the architecture dependencies. Our interceptor has been implemented for Linux and Solaris.

Other popular package managers like pkg [3] (used on Solaris systems), lpp [2] (used on AIX

systems), dpkg [1] (used on Debian Linux distributions) have interfaces similar to that of RPM and hence our approach is applicable to these package managers with the corresponding implementation changes that were described above. There are a few other package managers like SEPP [4], SLP [5] (used on Stampede Linux) that simply do not offer convenient interfaces to query packages and the package databases, and hence are not particularly suitable for our approach.

Conclusion

In this paper, we have discussed the design of a secure software installation framework. Our framework allows a system administrator to control the installation process through a configurable set of policies and enforce these policies through static and runtime checks. Our future work would include exporting the prototype in the context of other operating systems and package managers.

Author Biographies

All the authors of this paper are members of the Secure Systems Laboratory of Stony Brook and their homepages are accessible on the web from the laboratory page at <http://www.seclab.cs.sunysb.edu>.

R. Sekar is currently an Associate professor of Computer Science and heads the Secure Systems laboratory at SUNY, Stony Brook. Prof. Sekar's research interests include computer system and network security, software and distributed systems, programming languages and software engineering. He can be reached by electronic mail at sekar@cs.sunysb.edu.

VN Venkatakrishnan is a Ph.D. student in the Computer science department at Stony Brook. His main research area is computer security and is currently working on combining static and runtime techniques for assuring software security. He is available by email at venkat@cs.sunysb.edu.

Tapan Kamat is a M.S. student in the Computer Science department at Stony Brook. Tapan does research in the area of computer security. He can be reached via email at tkamat@cs.sunysb.edu.

Sofia Tsipa is currently working as a System Administrator in the Network Operations Center at the University of Thessaloniki, Greece after completing

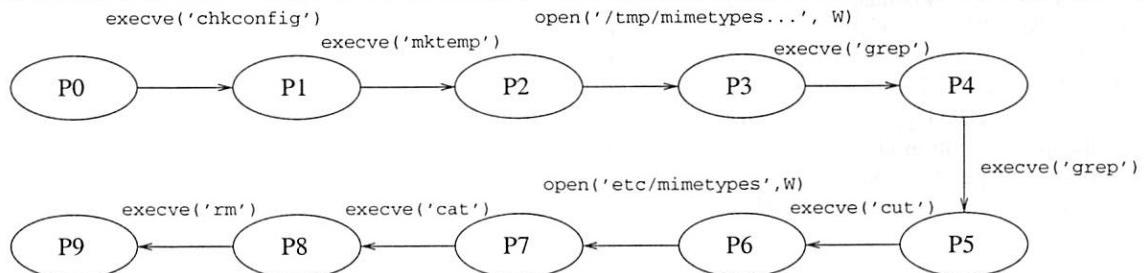


Figure 6: Model of apache server's post-install script.

her M.S. degree in the computer science department at SUNY, Stony Brook. She can be reached via email at sofia@cs.sunysb.edu.

Zhenkai Liang is a Ph.D. student in Computer Science at SUNY, Stony Brook. His research interests include computer security and algorithms. He can be reached by email at zliang@cs.sunysb.edu.

References

- [1] *dpkg: A Medium Level Package Manager for Debian*, <http://www.debian.org/doc/manuals/quick-reference/ch-package>.
- [2] *lpp: Aix package distribution*, http://usgibm.nersc.gov/doc_link/en_US/a_doc_lib/aixins/inslppkg/toc.htm.
- [3] *pkg: Solaris package distribution*, Solaris man pages.
- [4] *Sepp: Software Installation and Sharing System*, <http://www.sepp.ee.ethz.ch/seppdoc.pdf>.
- [5] *Slp: Stampede Linux Packages*, <http://www.marblehorse.org/projects/slps/SLPv5a-draft-specification.html>.
- [6] Anderson, E. and D. Patterson, “A Retrospective on Twelve Years of lisa Proceedings,” *Proceedings of Usenix System Administration, LISA*, 1999.
- [7] Mitchell, M. W. C. and P. Wild, *Contemporary Cryptology: The Science of Information Integrity*, Chapter Digital Signatures, IEEE Press, Piscataway, NJ, 1992.
- [8] Cousot, P., “Static Determination of Dynamic Properties of Programs,” *Proceedings of the Second International Symposium on Programming*, Paris, 1976.
- [9] Gong, L., M. Mueller, H. Prafullchandra, and R. Schemers, “Going Beyond the Sandbox: An Overview of the New Security Architecture In the Java Development Kit 1.2,” *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Monterey, California, December 1997.
- [10] Jain, K. and R. Sekar, “User-level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement,” *ISOC Network and Distributed System Security*, 2000.
- [11] Necula, G., “Proof Carrying Code,” *ACM Principles of Programming Languages*, 1997.
- [12] Sekar, R., C. Ramakrishnan, I. Ramakrishnan, and S. Smolka, “Model Carrying Code: A New Paradigm for Mobile Code Security,” *Proceedings of the New Security Paradigms Workshop*, 2001.
- [13] Sekar, R. and P. Uppuluri, “Synthesizing Fast Intrusion Prevention/Detection Systems From High-Level Specifications,” *Proceedings of the USENIX Security Symposium*, 1999.

Network-based Intrusion Detection – Modeling for a Larger Picture

Atsushi Totsuka – Tohoku University

Hidenari Ohwada – NTT, Tokyo

Nobuhisa Fujita – Tohoku University

Debasish Chakraborty – Tohoku University

Glenn Mansfield Keeni – Cyber Solutions, Inc.

Norio Shiratori – Tohoku University

ABSTRACT

The Internet is changing computing more than ever before. As the possibilities and the scopes are limitless, so too are the risks and chances of malicious intrusions. Due to the increased connectivity and the vast spectrum of financial possibilities, more and more systems are subject to attack by intruders. One of the commonly used method for intrusion detection is based on anomaly. Network based attacks may occur at various levels, from application to link levels. So the number of potential attackers or intruders are extremely large and thus it is almost impossible to “profile” entities and detect intrusions based on anomalies in host-based profiles. Based on meta-information, logical groupings has been made for the alerts that belongs to same logical network, to get a clearer and boarder view of the perpetrators. To reduce the effect of probably insignificant alerts a threshold technique is used.

Introduction

Intrusion detection today covers a wider scope than the name suggests. IDS systems are tasked to detect – reconnaissance, break-ins, disruption of services and attempts at any of these activities. IDS systems are also expected to identify the perpetrator or provide useful clues toward that end. Some IDS systems adopt defensive actions when faced with a (potential) attack. In classical host-based intrusion detection [2] – anomaly based detection techniques are employed. Anomaly is detected by comparing the profile or behavior of entities with their *normal* profiles. Profile is the pattern of actions of subjects on objects. The entity-space should be small enough to enable profiling of entities or groups of entities. The entity itself might be the perpetrator [or, is compromised by the perpetrator].

Network-based attacks which generally precede host break-ins do not lend themselves to easy profiling. An attack may occur at the link level, network level, transport level or at the application level. Thus the entities that are potential attackers or intruders are not just users with *user-ids* but link-level entities represented by MAC addresses, network level entities (represented by network addresses), transport level entities (represented by network address and transport protocol) or network application level entities (represented by network address, transport protocol, and port address). This leads to an explosion in the entity space making it all but impossible to “profile” entities and detect intrusions based on anomalies with respect to the profiles. Effectively, all communication –

packets or trains of packets need to be examined to detect traces of (attempted) mischief.

If (potential) mischief is detected, identifying the perpetrator is a hard task. The perpetrator is generally not directly related to the packet or datagram which is the only clue to the offender in the network context. The source address in the relevant IP datagram may vary over time for the same attack due to DHCP assignment or, due to deliberate maneuvers by the attacker. In one special case the source address may be spoofed altogether.

In a similar manner identifying the target of the attack may be difficult – the destinations may range over several ports of several hosts and over several networks. The target may be an application, a host, a network or networks of an organization, region or country. The perpetrator may be an application, a host, a network or even networks from a region or a country.

Added to the inherent difficulty in identifying the perpetrator and/or target is the fact that the rules or signatures that are employed to detect (potential) mischief are simplistic. Presence of the signatures do not necessarily signify mischief. Moreover, in the open Internet there are deliberate mischief makers making a concerted effort to break-in, the unwary user who (probably) unintentionally fires off a scan or mischievous program and malfunctioning programs that send off suspicious looking packets. The end result is a profusion of alerts from Intrusion detection systems. When looked at in isolation the alerts make little sense, and serve little more than log messages destined for posterity.

To provide greater visibility of (potential) attacks, perpetrators and targets, we have devised a

model that aggregate entities and actions into logical super-entities and actions. Thus a set of host-IP addresses, get clubbed into a logical network. And attacks from this logical network constitute a larger attack. To reduce the noise of (probably) irrelevant alerts that effectively hinder the identification of actual offenses and offenders, a thresholding technique is used. Experimental results shows the effect of our model, which is based on logical groupings and threshold techniques.

In the next section, we will discuss about the background and related works. Then we will talk about our proposed model of Network-based Intrusion Detection and subsequently about the observation environment of our case study and its evaluation. We then conclude our work.

Background

It is very important that the security mechanism of a system are designed so as to prevent unauthorized access to system resources and data. The conventional approach to secure a computer or network system is to build a protective shield around it. External users must identify themselves to enter the system. This shield should prevent leakage of information from the protected domain to the outside world. But it is not possible to design a system which is completely secure. We can however try to detect these intrusion attempts so that action may be taken to repair the damage and also the identification of the perpetrators and their victims. If there are attacks on a system we would like to detect them as soon as possible, preferably in real-time and take preventive measure. This is essentially what an Intrusion Detection System (IDS) does.

Techniques of intrusion detection can be divided into mainly two types. **Anomaly based detection** and **Signature based detection**. Anomaly detection techniques assume that all intrusive activities are necessarily anomalous. This means if we could establish a *normal activity profile* for a system, we could, in theory at least flag all system states deviating from the established profile by statistically significant amounts as intrusion attempts. But there are two possibilities, (i) anomalous activities that are not intrusive are flagged as intrusive (false positive) and (ii) anomalous activities that actually intrusive but not flagged (false negative). The second one is obviously more dangerous.

The main issues in anomaly detection systems thus become the selection of threshold levels so that neither of the above two problems is unreasonably magnified. The concept behind *signature detection* schemes is that there are ways to represent attacks in the form of a pattern or a signature so that even variations of the same attack can be detected. So in some sense, they are like virus detection systems. Able to detect known attack patterns but of little use in case of unknown attack methods. The main issues in Signature detection systems are how to write a signature

that encompasses all possible variations of the pertinent attack, and how to write signatures that do not also match non-intrusive activity. An interesting difference between these two schemes is that *anomaly detection* systems try to detect the complement of “bad” behavior, whereas *signature detection* system try to recognize known “bad” behavior.

There are advantages and disadvantages for both of the detection approaches.

- **Anomaly detection:**

- *Advantages:* No need to configure the system. It automatically learns the behavior of a large number of subjects, and can be left to run unattended. It has the possibilities of catching novel intrusions, as well as variations of known intrusions.

- *Disadvantages:* It only flags unusual behavior, not necessarily illicit one. It could pose a problem when two types of behavior do not overlap. A system will not find anything wrong with a particular user, who changes his behavior slowly before attack. Updating of subject's profiles, and the correlation of current behavior with those profiles is typically a computationally intensive task, that can be too heavy for the available resources.

- **Signature detection:**

- *Advantages:* The system knows for a fact which is suspicious behavior and which is not. This is a simple and efficient processing of the audit data. The rate of false positive can also be kept low.

- *Disadvantages:* Specifying the detection signatures is a highly qualified, and time consuming task. It is not something that “ordinary” operators of the system would do. Depending on how the signatures are specified, subtle variations of the intrusion scenarios can lead to them going undetected.

Early work on intrusion detection was due to Anderson [1] and Denning [2]. Since then, it has become a very active field. Most intrusion detection system (IDS) are based on one of two methodologies: either they generate a model of a program's or system's behavior from observing its behavior on known inputs [9], or they require the generation of a rule base [8]. A detailed discussion on network intrusion detection can be found in [6, 7].

Host based intrusion detection to network based detection correlates with the shift from single multi-user systems to network of workstations. As computers and networks get faster, we can process more audit data per unit time, but that same computer or network unfortunately produce audit data at a much higher rate as well. Hence the total ration of consumed resources to

available resources is, if not constant, at least not decreasing at a sufficiently fast pace, that the performance of the intrusion detection system becomes a non-issue. The amount of data that need to be processed remains as a vital problem for intrusion detection. So it becomes much more difficult to detect network based attacks than host-based attacks. There is still lack of study in the field of coverage, of the intrusions the system can realistically be thought to handle. The problems are both that of incorrectly classifying benign activity as intrusive and called *false positive*, and that of classifying intrusive activity as not-intrusive, as *false negative*. These mis-classifications lead to different problem. We tried to cover both the issues in this paper.

Network-based Intrusion Detection Model

To provide greater visibility of (potential) attacks, perpetrators and targets we have devised a model that aggregate entities and actions into logical super-entities and actions. Thus a set of host-IP addresses, get clubbed into a logical network. And attacks from this logical network constitute a larger attack. To reduce the noise of (probably) irrelevant alerts that effectively hinder the identification of actual offenses and offenders a thresholding technique is used.

The logical groupings are carried out based on what we call meta-information or "glue" information. This meta-information is in effect a pool of "hints" which indicate how the pieces of a puzzle posed by the alerts (may) fit together to form a larger picture. Its contents are network topological information, organizational network information, *Autonomous System* (AS) information, routing information, *Domain Name System* information, geopolitical information. Most of these components are readily available in the network.

The threshold is a tunable parameter. It can be varied to provide the best visibility.

The larger picture

The isolated incidents reported as alerts when seen in the context of the meta-information form a clearer picture. The application, server and/or network that is being targeted becomes clear, the source of the attack gets amplified and the relation between scans and subsequent attacks begin to emerge. The application of thresholds to filter out the noise makes the patterns even clearer. The significant effect is that we have a much clearer view of the perpetrator, and a much deeper understanding of the target of the attack.

We have carried out case studies on several operational networks and verified the effectiveness of the approach.

Case study

We have carried out a case study by observing the alerts generated on three networks. The first, observation point 1, is a network connecting 10 computers, the second, observation point 2 is a network

connecting approximately 30 computers, the third, observation point 3, connects a large scale campus network to the Internet.

We used Snort [3] to detect suspicious traffic. We used about 1200 signatures. The profiles of the potential attacks were observed for 183 days. As the meta-information we used the IP-address to AS-number mapping available from the routing registry [4], organization to network address mapping using the DNS system, and organization to country mapping using a locally compiled database (with network sources as input).

Evaluation

Clarity From Aggregation of Profiles

Three separate modes of aggregation are possible – source based, destination based and source-destination based. We experimented with all three modes and found source based aggregation to be the least effective. This is expected as more often than not the source addresses are spoofed. The effect of aggregation is most pronounced when destination based aggregation is carried out.

The advantages of aggregation is twofold. First, it can reduce the total amount of data by less than half. So for analyzing the data, irrespective of either doing it by manually or not, it will be much more easier to handle if the volume of data is considerably less. Secondly, aggregation will give a much more clearer view of the attacker(s) and also about the victim(s). If we see the attacks individually it may look like they are coming from different places without any relation between them. But in aggregation, we can find whether those attacks are generated from the same network entity or not. Thus it will be easier to locate a perpetrator. Similarly, for destination (or victims of an attack) it will be easier to identify the actual target. For example, in case of port-scanning for a particular destination, apparently it may look different, but in aggregation we can find which destination the attacker is targeting. Because in both the cases aggregation can give amplified picture of the source and destination of an attack.

We then compared number of profiles generated using conventional methods and proposed methods. We define a rate of aggregation as:

$$\text{Aggreg. Rate} = 100\% \times \frac{\# \text{ proposed model profiles}}{\# \text{ conventional model profiles}}$$

Table 1 shows the total numbers of profiles for the 183 days as seen in the data at the three observation points. The data for the source-destination based aggregation is given here. From this table it is clear that due to aggregation the amount of data has been reduced to less than half indicating greater feasibility of analysis. Not only that, it will also bring clarity. The aggregation scheme enables one to detect types of attacks which could not be detected otherwise. If source IP addresses are aggregated, an attack from distributed sources in a single network would be more likely to be detected. While, if destination addresses are aggregated, an attack which apparently looks like aimed at different individual

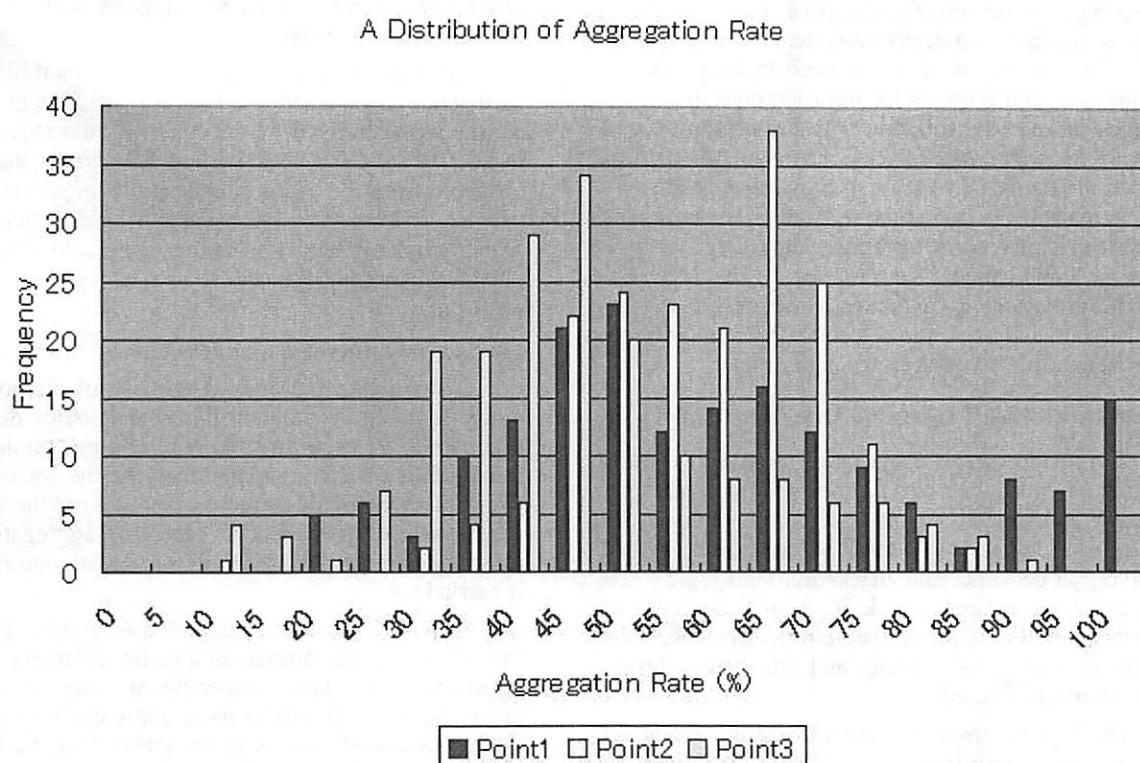


Figure 2: Distribution of the rate of aggregation of profile.

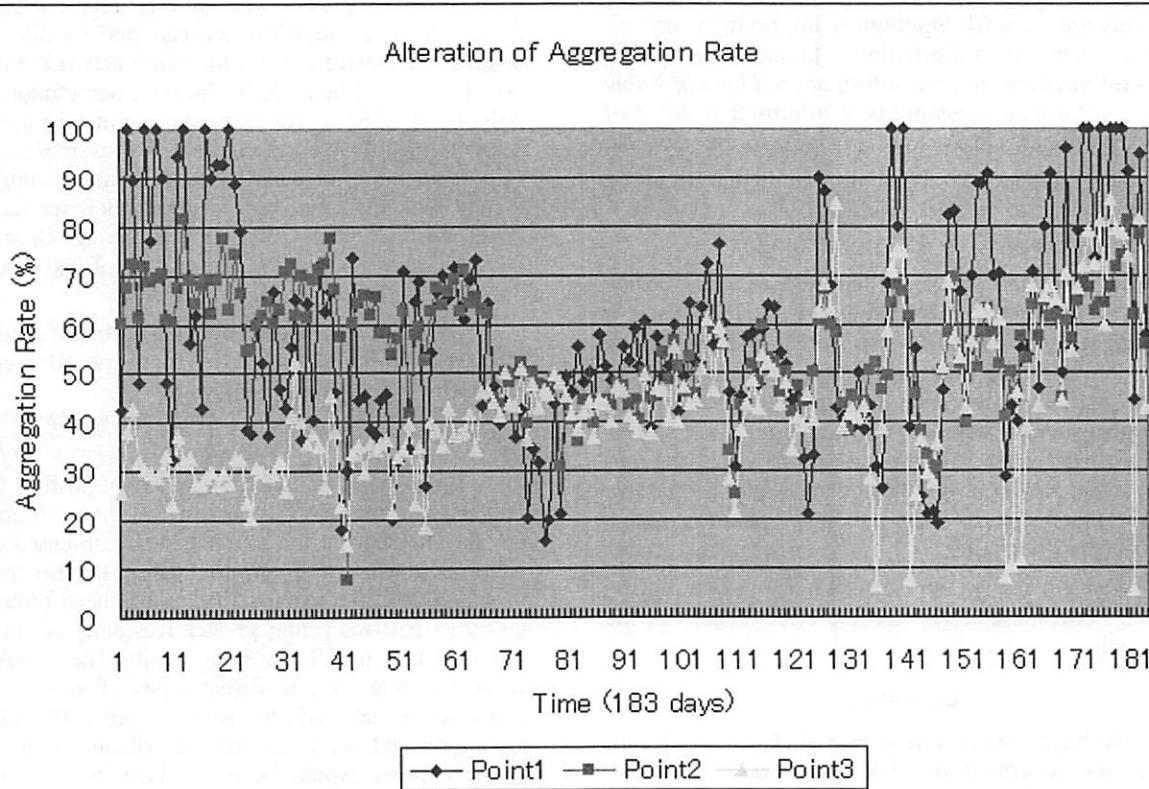


Figure 3: Change of the rate of aggregation during the period of observation.

destination without any relation, may found to be multiple destinations of the same network.

Figure 2 shows the distribution of the rates of aggregation per day, where the horizontal axis is the rate of aggregation and the vertical axis is the frequency (the number of days). Figure 3 shows the change of the rate of aggregation during the period of observation. The horizontal axis is time (183 days) and the vertical axis is the rate of aggregation.

Simple AS based aggregation reduces the number of profiles by a factor of approximately 50%. The result at observation point 3 shows that such aggregation is more effective when Internet traffic is involved.

We see that the aggregation varies for almost everyday as shown in Figures 2 and 3. There are days with no aggregation, and days when aggregation rate is very high.

	Conventional Model	Proposed Model	Aggreg. Rate
Point 1	10443	4971	47.60%
Point 2	61459	30030	48.86%
Point 3	348707	113221	32.46%

Table 1: Clarity from aggregation of profiles.

Another important observation is that the number of rejected alerts when thresholding technique is used in conjunction with the logical aggregation is much smaller than that when thresholding is used in isolation. This is significant as it implies that the results are safer and more accurate.

In Figure 4 we have shown the effect of threshold for both IP based and AS based alerts. Here

horizontal axis is the Threshold values and vertical axis is the percentage of number of alerts that has been covered (after ignoring the below-threshold value alerts). ‘SIP’ and ‘DIP’ means the ‘Source IP’ and ‘Destination IP’ respectively. Similarly, ‘SAS’ and ‘DAS’ means ‘Source AS’ and ‘Destination AS.’

Obviously when the threshold value is zero, when there is no rejection of alerts, there is no chance of any ‘false negative.’ Because at that time we are considering every single alerts into our account. Increase in threshold value (increase the rejected alerts) affect the IP based alerts much more than AS based alerts. It implies that AS based grouping is more effective for getting a more clearer picture of the attacker and the victim as well, as it is covering a greater range. At the same time it is also an indicator that in our approach the chance of ‘false negative’ is very low. Because even after increasing the threshold value to a much higher degree, the total number of rejected alerts are comparatively lower than that of conventional IP-based approach. And in our model, even if it may not reduce but there is no scope of increasing ‘false positive’ alerts than conventional models.

Conclusion

The aggregation technique envisaged in the model helps in providing much greater clarity to the results. When used in conjunction with the thresholding technique its effect is very significant. Results from the case studies show that it is possible to reduce the number of entities that require close attention, to a manageable small set. We have also found that in our approach the chance of rejecting actual intrusive alerts (false negative) is much less, which is considered as a

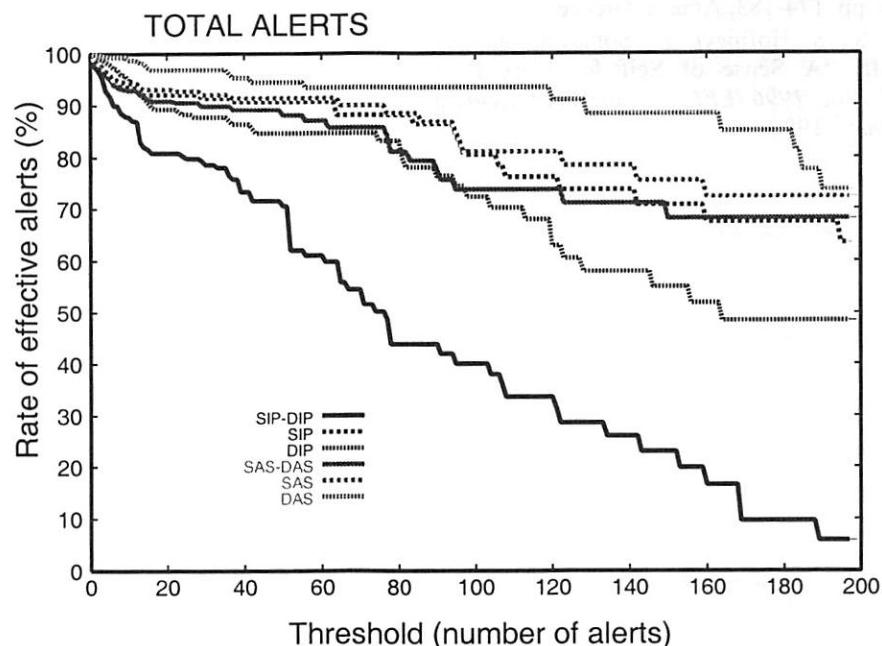


Figure 4: Percentage of number of alerts for both IP-based and AS-based profiles with different threshold values.

far more serious problem than the problem of false positive.

These techniques coupled with network configuration information and network visualization techniques [5] are likely to have a significant impact on intrusion detection systems.

References

- [1] Anderson, J. P., "Computer Security Threat Monitoring and Surveillance," Technical report, James P. Anderson Company, For Washington, Pennsylvania, April 1980.
- [2] Denning, Dorothy E., "An Intrusion Detection Model," *IEEE Transaction on Software Engineering*, Vol. SE-13, No.2, February 1987, 222-232.
- [3] Roesch, M., "Snort: Lightweight intrusion detection for networks," *USENIX LISA'99*, <http://www.snort.org/>, November 1999.
- [4] IRRd, *Internet Routing Registry Daemon*, <http://www.irrd.net/>.
- [5] Mansfield, Glenn, et al., "Towards Trapping Wily Intruders in the Large," *Computer Networks*, Vol 34, Issue 4, October 2000.
- [6] Banerjee, Biswanath, L. Todd Heberlein, and Karl N. Levitt, "Network Intrusion Detection," *IEEE Network*, May/June, 1994.
- [7] Axelsson, Stefan, "Research in Intrusion-Detection Systems: A Survey," TR: 98-17, Revised August 19, 1999.
- [8] Bernaschi, M., E. Gabrielli, and L. V. Mancini, "Operating System Enhancements to Prevent the Misuse of System Calls," *Proc. of the 7th ACM Conference on Computer and Communications Security*, pp. 174-183, Athens, Greece.
- [9] Forrest, S., S. Hofmeyr, A. Somayaji, and T. Longstaff, "A Sense of Self for Unix Processes," *Proc. 1996 IEEE Symposium of Security and Privacy*, 1996.

Timing the Application of Security Patches for Optimal Uptime

Steve Beattie, Seth Arnold, Crispin Cowan, Perry Wagle, and Chris Wright[†]

– WireX Communications, Inc.

Adam Shostack – Zero Knowledge Systems, Inc.

ABSTRACT

Security vulnerabilities are discovered, become publicly known, get exploited by attackers, and patches come out. When should one apply security patches? Patch too soon, and you may suffer from instability induced by bugs in the patches. Patch too late, and you get hacked by attackers exploiting the vulnerability. We explore the factors affecting when it is best to apply security patches, providing both mathematical models of the factors affecting when to patch, and collecting empirical data to give the model practical value. We conclude with a model that we hope will help provide a formal foundation for when the practitioner should apply security updates.

Introduction

*"To patch, or not to patch, – that is the question: –
Whether 'tis nobler in the mind to suffer
The slings and arrows of outrageous script kiddies,
Or to take up patches against a sea of troubles,
And by opposing, end them?" [24]*

"When to patch?" presents a serious problem to the security administrator because there are powerful competing forces that pressure the administrator to apply patches as soon as possible and also to delay patching the system until there is assurance that the patch is not more likely to cause damage than it proposes to prevent. Patch too early, and one might be applying a broken patch that will actually cripple the system's functionality. Patch too late, and one is at risk from penetration by an attacker exploiting a hole that is publicly known. Balancing these factors is problematic.

The pressure to immediately patch grows with time after the patch is released, as more and more script kiddies acquire scanning and attack scripts to facilitate massive attacks [4]. Conversely, the pressure to be cautious and delay patching decreases with time, as more and more users across the Internet apply the patch, providing either evidence that the patch is defective, or (through lack of evidence to the contrary) that the patch is likely okay to apply. Since these trends go in opposite directions, it should be possible to choose a time to patch that is optimal with respect to the risk of compromising system availability. Figure 1 conceptually illustrates this effect; where the lines cross is the optimal time to patch, because it minimizes the total risk of loss.

This paper presents a proposed model for finding the appropriate time to apply security patches. Our approach is to model the cost (risk and consequences)

of penetration due to attack and of corruption due to a defective patch, with respect to time, and then solve for the intersection of these two functions.

These costs are functions of more than just time. We attempt to empirically inform the cost of failure due to defective patches with a survey of security advisories. Informing the cost of security penetration due to failure to patch is considerably more difficult, because it depends heavily on many local factors. While we present a model for penetration costs, it is up to the local administrator to determine this cost.

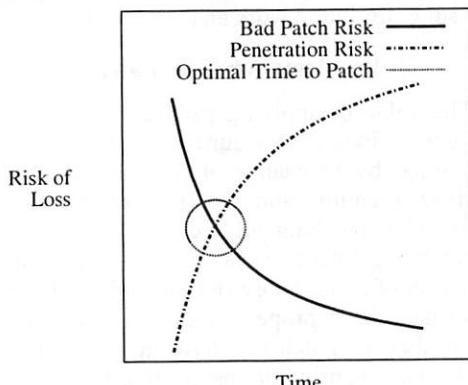


Figure 1: A hypothetical graph of risks of loss from penetration and from application of a bad patch. The optimal time to apply a patch is where the risk lines cross.

In particular, many security administrators feel that it is imperative to patch vulnerable systems *immediately*. This is just an end-point in our model, representing those sites that have very high risk of penetration and have ample resources to do local patch testing in aid of immediate deployment. Our intent in this study is to provide guidelines to those who do not have sufficient resources to immediately test and patch everything, and must choose where to allocate scarce

[†]This work supported by DARPA Contract F30602-01-C-0172.

security resources. We have used the empirical data to arrive at concrete recommendations for when patches should be applied, with respect to the apparent common cases in our sample data.

It should also be noted that we are *not* considering the issue of when to disable a service due to a vulnerability. Our model considers only the question of when to patch services that the site must continue to offer. In our view, if one can afford to disable a service when there is a security update available, then one probably should not be running that service at all, or should be running it in a context where intrusion is not critical.

Lastly, we do not believe that this work is the final say in the matter, but rather continues to open a new area for exploration, following on Browne, et al. [4]. As long as frequent patching consume a significant fraction of security resources, resource allocation decisions will have to be made concerning how to deal with these patches.

The rest of this paper is structured as follows. The next section presents motivations for the models we use to describe the factors that make patching urgent and that motivate caution. Then, the next section formally models these factors in mathematical terms and presents equations that express the optimal time to apply patches. The subsequent section presents methods and issues for acquiring data to model patch failure rates. The paper then presents the empirical data we have collected from the Common Vulnerabilities and Exposures (CVE) database and describes work related to this study. The paper ends with discussions the implications of this study for future work and our conclusions.

Problem: When To Patch

The value of applying patches for known security issues is obvious. A security issue that will shortly be exploited by thousands of script-kiddies requires immediate attention, and security experts have long recommended patching all security problems. However, applying patches is not free: it takes time and carries a set of risks. Those risks include that the patch will not have been properly tested, leading to loss of stability; that the patch will have unexpected interaction with local configurations, leading to loss of functionality; that the patch will not fix the security problem at hand, wasting the system administrator's time. Issues of loss of stability and unexpected interaction have a direct and measurable cost in terms of time spent to address them. To date, those issues have not been a focus of security research. There is a related issue: finding a list of patches is a slow and labor-intensive process [7]. While this makes timely application of patches less likely because of the investment of time in finding them, it does not directly interact with the risk that applying the patch will break things. However, the ease of finding and applying patches has begun to get substantial public attention [20] and is not our focus here.

Most system administrators understand that these risks are present, either from personal experience or from contact with colleagues. However, we know of no objective assessment of how serious or prevalent these flaws are. Without such an assessment it is hard to judge when (or even if) to apply a patch. Systems administrators have thus had a tendency to delay the application of patches because the costs of applying patches are obvious, well known, and have been hard to balance against the cost of not applying patches. Other sources of delay in the application of patches can be rigorous testing and roll-out procedures and regulations by organizations such as the US Food and Drug Administration that require known configurations of systems when certified for certain medical purposes [1].

Some organizations have strong processes for triaging, testing, and rolling-out patches. Others have mandatory policies for patching immediately on the release of a patch. Those processes are very useful to them, and less obviously, to others, when they report bugs in patches. The suggestions that we make regarding delay should not be taken as a recommendation to abandon those practices.¹

The practical delay is difficult to measure, but its existence can be inferred from the success of worms such as Code Red. This is illustrative of the issue created by delayed patching, which is that systems remain vulnerable to attack. Systems which remain vulnerable run a substantial risk of attacks against them succeeding. One research project found that systems containing months-old known vulnerabilities with available but unapplied patches exposed to the Internet have a "life expectancy" measured in days [14]. Once a break-in has occurred, it will need to be cleaned up. The cost of such clean-up can be enormous.

Having demonstrated that all costs relating to patch application can be examined in the "currency" of system administrator time, we proceed to examine the relationship more precisely.

Solution: Optimize the Time to Patch

To determine the appropriate time to patch, we need to develop a mathematical model of the potential costs involved in patching and not patching at a given time. In this section we will develop cost functions that systems administrators can use to help determine an appropriate course of action.

First, we define some terms that we will need to take into account:

- e_{patch} is the expense of fixing the problem (applying the patch), which is either an opportunity cost, or the cost of additional staff.
- $e_{p.recover}$ is the expense of recovering from a failed patch, including opportunity cost of work

¹As a perhaps amusing aside, if everyone were to follow our suggested delay practice, it would become much less effective. Fortunately, we have no expectation that everyone will listen to us.

delayed. Both this and the next cost may include a cost of lost business.

- e_{breach} is the expense of recovering from a security breach, including opportunity cost of work delayed and the cost of forensics work.
- p_{fail} is the likelihood that applying a given patch will cause a failure.
- p_{breach} is the likelihood that not applying a given patch will result in a security breach.

All of these costs and probabilities are parameterized. The costs e_{patch} , $e_{p.recover}$, and e_{breach} are all particular to both the patch in question and the configuration of the machine being patched. However, because the factors affecting these costs are so specific to an organization, we treat the costs as constants. This is constant *within* an organization, not between organizations, which we believe is sensible for a given systems administrator making a decision.

The probabilities p_{fail} and p_{breach} vary with time. Whether a patch is bad or not is actually a fixed fact at the time the patch is issued, but that fact only becomes known as the Internet community gains experience applying and using the patch. So as a patch ages without issues arising, the probability of a patch *turning out to be* bad decreases.

The probability p_{breach} is a true probability that increases with time in the near term. Browne, et al. [4] examined exploitation rates of vulnerabilities and determined influencing terms such as the release of a scripted attack tool in rates of breaches. However, the rate of breach is not a simple function $\frac{1}{|InternetHosts|}$

or even $\frac{N}{|InternetHosts|}$ (where N is the number of hosts or unprotected hosts that a systems administrator is responsible for and $|InternetHosts|$ is the number of hosts on the Internet). Not every host with a vulnerability will be attacked, although in the wake of real world events such as the spread of Code Red [6] and its variants, as well as work on Flash [26] and Warhol [28] worms, it seems that it may be fair to make that assumption.

Thus we will consider both probabilities p_{fail} and p_{breach} as functions of time (t), and write them $p_{fail}(t)$ and $p_{breach}(t)$.

Next, we want to develop two cost functions:

- $c_{patch}(t)$: cost of patching at a given time t .
- $c_{npatch}(t)$: cost of not patching at a given time t .

The probable cost of patching a system drops over time as the Internet community grows confidence in the patch through experience. Conversely, the probable cost of not patching follows a ‘ballistic’ trajectory, as the vulnerability becomes more widely known, exploitation tools become available, and then fall out of fashion [4]; but, for the part of the ballistic curve we are concerned with, we can just consider cost of not patching to be monotonically rising. Therefore, the

administrator will want to patch vulnerable systems at the earliest point in time where $c_{patch}(t) \leq c_{npatch}(t)$.

The cost of patching a system will have two terms: the expense of applying the patch, and the expense of recovering from a failed patch. Applying a patch will likely have a fixed cost that must be paid regardless of the quality of the patch. Recovery cost, however, will only exist if a given patch is bad, so we need to consider the expected risk in a patch. Since a systems administrator cannot easily know a priori whether a patch is bad or not, we multiply the probability that the patch induces failure by the expected recovery expense. This gives us the function

$$c_{patch}(t) = p_{fail}(t)e_{p.recover} + e_{patch} \quad (1)$$

It is possible, although not inexpensive, to obtain much better estimations of the probability of failure through the use of various testing mechanisms, such as having a non-production mirror of the system, patching it, and running a set of tests to verify functionality. However, such systems are not the focus of our work.

The cost of not applying a patch we consider to be the expense of recovering from a security breach. Again, an administrator is not going to know a priori that a breach will occur, so we consider the cost of recovery in terms of the probability of a security breach occurring. Thus we have:

$$c_{npatch}(t) = p_{breach}(t)e_{breach} \quad (2)$$

Pulling both functions together, a systems administrator will want to patch vulnerable systems when the following is true:

$$p_{fail}(t)e_{p.recover} + e_{patch} \leq p_{breach}(t)e_{breach} \quad (3)$$

In attempting to apply the functions derived above, a systems administrator may want to take more precise estimates of various terms.

On the Cost Functions

Expenses for recovering from bad patches and security breaches are obviously site and incident specific, and we have simplified some of that out to ease our initial analysis and aid in its understanding.

We could argue with some confidence that the cost of penetration recovery often approximates the cost of bad patch recovery. In many instances, it probably amounts to “reinstall.” This simplifying assumption may or may not be satisfactory. Recovery from a break-in is likely harder than recovery from a bad patch, because recovery from bad patch may simply be a reinstall, or at least does not involve the cost of dealing with malice, while recovery from getting hacked is identifying and saving critical state with tweezers, reformatting, re-installation, applying patches, recovering state from backup, patching some more, ensuring that the recovered state carries no security risk, and performing forensics, a non-trivial expense [10]. However, it is possible that recovery from a bad patch could have a higher cost than penetration recovery – consider a patch that introduces subtle file system corruption that is not detected for a year.

Furthermore, we note that many vendors are working to make the application of security patches as simple as possible, thereby reducing the expense of applying a security patch [22, 25, 29]. As the fixed cost of applying a patch approaches zero, we can simply remove it from the equations:

$$p_{fail}(t) e_{p.recover} \leq p_{breach}(t) e_{breach} \quad (4)$$

Alternately, we can assume that recovery from being hacked is C times harder than recovery from bad patch (C may be less than one). While the math is still fairly simple, we are not aware of systemic research into the cost of recovering from security break-ins. However, a precise formulation of the time is less important to this paper than the idea that the time absorbed by script kiddies can be evaluated as a function of system administrator time. Expenses incurred in recovery are going to be related to installation size and number of affected machines, so an argument that there is some relationship between costs can be made. This allows us to state that:

$$e_{breach} = C e_{p.recover} \quad (5)$$

We can substitute this into equation 4:

$$p_{fail}(t) e_{p.recover} \leq p_{breach}(t) C e_{p.recover} \quad (6)$$

Dividing each side by $e_{p.recover}$, we arrive at the decision algorithm:

$$p_{fail}(t) \leq p_{breach}(t) C \quad (7)$$

Recall our assumptions that $p_{breach}(t)$ rises with time and $p_{fail}(t)$ drops with time. Therefore, the earliest time t that equation 7 is satisfied is the optimal time to apply the patch.

When to Start the Clock

While we discuss the value of the equations above at given times, there are actually numerous points from which time can be counted. There is the time from the discovery of a vulnerability, time from the public announcement of that vulnerability, and time since a patch has been released. Browne, et al. [4] work from the second, since the first may be unknown, but the spread of the vulnerability information may be better modeled from the first, especially if the vulnerability is discovered by a black hat. A systems administrator may only care from the time a patch is available, although some may choose to shut off services known to be vulnerable before that as a last resort, and work has been done on using tools such as chroot(2), Janus [12], and SubDomain [9, 13] to protect services that are under attack. In this paper, we have chosen to start counting time from when the patch is released.

Methodology

The first thing to consider when deciding to experimentally test the equations derived previously is a source of data. We considered starting with specific vendors' advisories. Starting from vendor data has flaws: it is difficult to be sure that a vendor has produced advisories for all vulnerabilities, the advisories may not link to other information in useful ways, and

different vendors provide very different levels of information in their advisories.

We decided instead to work from the Common Vulnerabilities and Exposures (CVE) [16], a MITRE-hosted project, to provide common naming and concordance among vulnerabilities. Since MITRE is an organization independent of vendors, using the CVE database reduces the chance of bias. Starting from CVE allows us to create generic numbers, which are useful because many vendors do not have a sufficient history of security fixes. However, there are also many vendors who do have such a history and sufficient process (or claims thereof) that it would be possible to examine their patches, and come up with numbers that apply specifically to them.

Data Gathering

Starting from the latest CVE (version 20020625), we split the entries into digestible chunks. Each section was assigned to a person who examined each of the references. Some of the references were unavailable, in which case they were ignored or tracked down using a search engine. They were ignored if the issue was one with many references (e.g., CVE-2001-0414 has 22 references, and the two referring to SCO are not easily found.) If there was no apparent patch re-issue, we noted that. If there was, we noted how long it was until the patch was withdrawn and re-released. Some advisories did not make clear when or if a bad patch was withdrawn, and in that case, we treated it as if it was withdrawn by replacement on the day of re-issuance.

Methodological Issues

*"There are more things in Heaven and Earth,
Horatio,
Then are dream't of in our Philosophy."* [24]

Research into vulnerabilities has an unfortunate tendency to confound researchers with a plethora of data gathering issues. These issues will impact the assessment of how likely a patch is to fail. It is important to choose a method and follow it consistently for the results to have any meaning; unfortunately, any method chosen causes us to encounter issues which are difficult and troubling to resolve. Once we select a method and follow it, our estimates may be systematically wrong for several reasons. Cardinality issues are among the worst offenders:

- **Vendors rolling several issues into one patch:** An example of this is found in one vendor patch [19] which is referenced by seven candidates and entries in the CVE (CAN-2001-0349, CAN-2001-0350, CVE-2001-0345, CVE-2001-0346, CVE-2001-0348, CVE-2001-0351, and CVE-2001-0347).
- **Vendors rolling one patch into several advisories:** An example here is CVE-2001-0414 with a dozen vendors involved. The vulnerability is not independent because Linux and BSD vendors commonly share fixes, and so an

update from multiple vendors may be the same patch. If this patch is bad, then in producing a generic recommendation of when to patch, we could choose to count it as N bad patches, which would lead to a higher value for p_{fail} , and consequently, later patching. If the single patch is good, then counting it as N good patches could bias the probability of patch failure downward.

- **Vendors releasing advisories with work-arounds, but no patches:** An example is CVE-2001-0221, where the FreeBSD team issued the statement “[this program] is scheduled for removal from the ports system if it has not been audited and fixed within one month of discovery.” No one fixed it, so no patch was released. A related situation occurred when a third party, unrelated to Oracle, released an advisory relating to Oracle’s product along with a workaround, and Oracle remained completely silent about the issue (CVE-2001-0326). We recorded these instances but treated them as non-events – our goal is to measure quality of patches; if no patch was released, there is nothing to measure.

There are several other potential sources of bias. We may not have accurate information on whether a vendor released an updated patch, because the CVE entry points to the original, and the vendor released a subsequent/different advisory. This potentially introduces a bias by reducing our computed probability of a harmful patch.

When patches are not independent, there is bias in a different direction; consider if one or more vendors released a revised update while others did not (for example, CVE-2001-0318). We considered each CVE entry as one patch, even if it involved multiple vendors. We chose to record the data for the vendor who issued the latest advisory revision (e.g., Debian over Mandrake and Conectiva in CVE-2001-0318). This potentially introduces a bias towards patches being less reliable than they actually are. Systems administrators tracking the advisories of one specific vendor would not have this potential source of bias.

It may be difficult to decide if a patch is bad or not. For example, the Microsoft patch for CVE-2001-0016 was updated six months after its release. There was a conflict between this patch and Service Pack 2 for Windows 2000. Installing the patch would disable many of the updates in Service Pack 2. Note that SP2 was issued four months after the patch, so there was four months where the patch was harmless, and two months where the patch and Service Pack 2 conflicted. We treated it as if was bad for the entire six months.

There is a potential for concern with the number of CVE entries we have examined. In the next section, we attempt to infer appropriate times to apply patches

by observing the knees in the curves shown in Figures 7 and 9, and these inferences would be stronger if there were sufficient data points to be confident that the knees were not artifacts of our sample data.

More data points would be desirable, but obtaining it is problematic. We found the CVE repository to be limiting, in that it was difficult to determine whether any given security patch was defective. For future research, we recommend using security advisory information direct from vendors. In addition to providing more detail, such an approach would help facilitate computing patch failure rate with respect to each vendor.

We do not believe that these issues prevent a researcher from analyzing the best time to patch, or a systems administrator from making intelligent choices about when to patch. However, these methodological issues do need to be considered in further studies of security patch quality.

Empirical Data

In this section, we examine the data we collected as discussed in the previous section. We examined 136 CVE entries, dating from 1999, 2000, and 2001. Of these, 92 patches never were revised leading us to believe they were safe to apply, 20 patches either were updated or pulled, and 24 CVE entries were non-patch events as discussed in ‘Methodological Issues.’ Table 2 summarizes this data. Of the 20 patches that were determined to be faulty, all but one (CVE-2001-0341) had an updated patch released. Of these, three were found to be faulty and had a second update released; one subsequently had a third revision released. Table 3 summarizes the data for the revised patches.

Total CVE entries examined	136
Good patches	92
Revised or pulled patches	20
Non-patch entries	24

Table 2: Quality of initial patches.

Revised or pulled patches	20
Good revised patches	16
Re-revised patches	3
Pulled and never re-released patches	1

Table 3: Quality of revised patches.

Table 4 analyzes the properties of the patch revisions. The middle column shows the number of days from the initial patch release until an announcement of some kind appeared indicating that the patch was bad, for the 20 patches that were revised. The right column shows the number of days from the revised patch release until notification that the revised patch was faulty, for the three issues that had subsequent revisions. Three data points is insufficient to draw

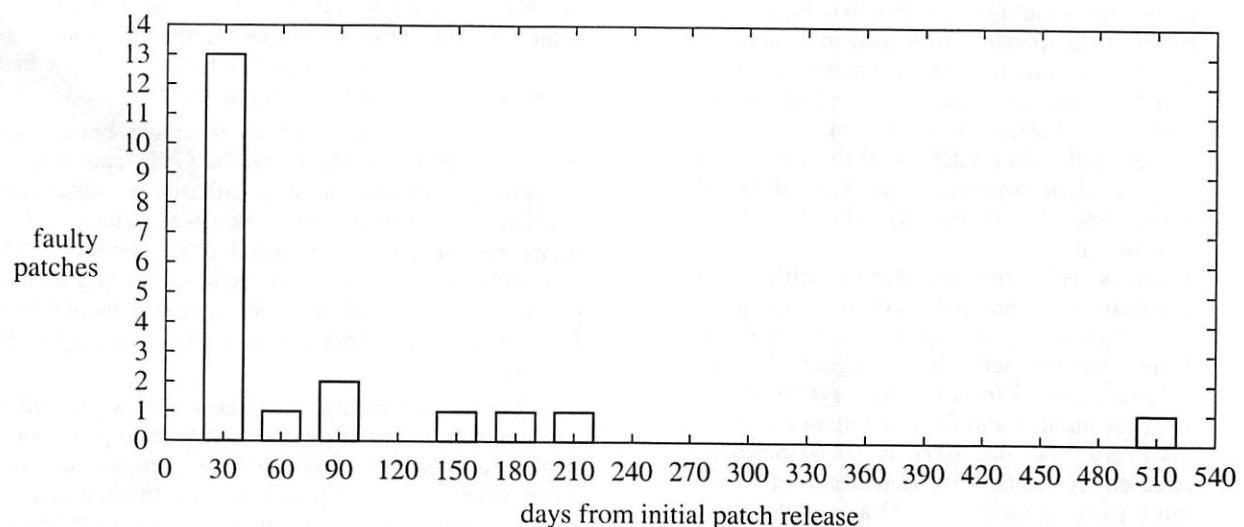


Figure 5: A histogram of the number of faulty initial patches.

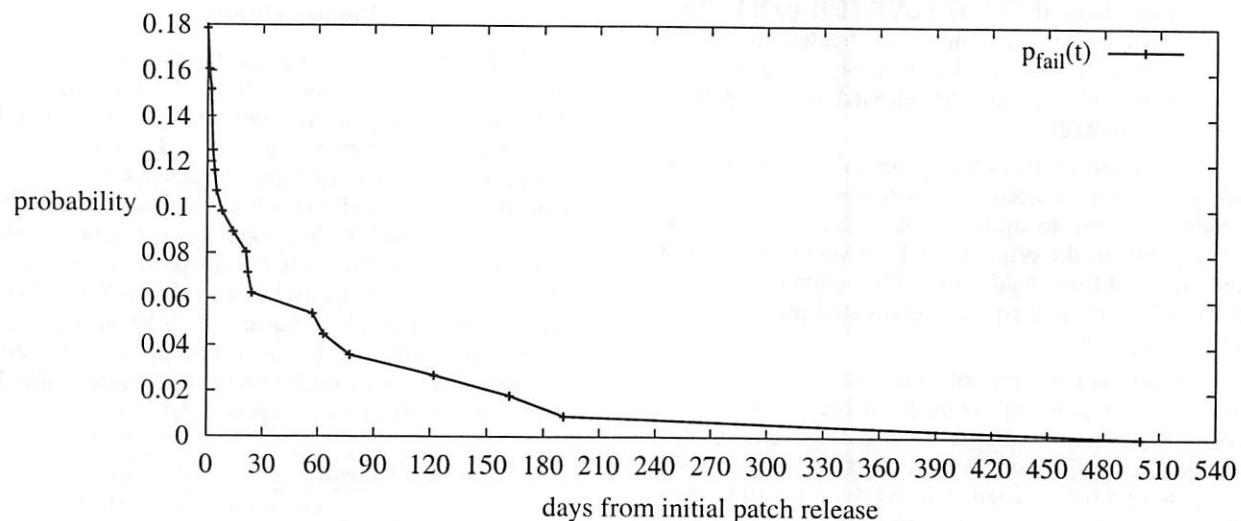
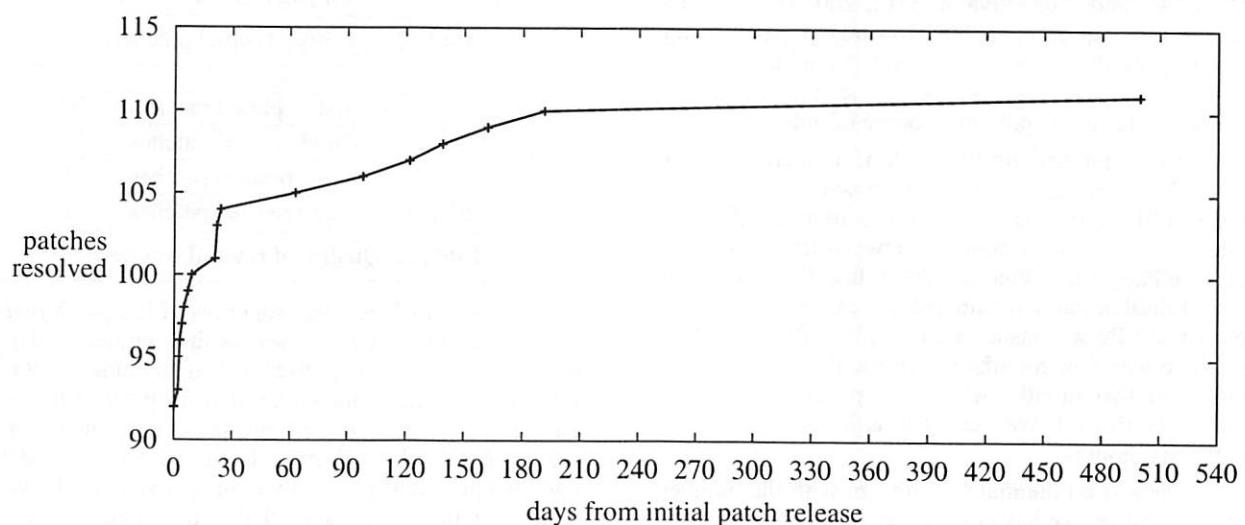
Figure 6: The probability $p_{fail}(t)$ that an initial patch has been incorrectly identified as safe to apply.

Figure 7: A cumulative graph showing the time to resolve all issues.

meaningful conclusions, so we will disregard doubly or more revised patches from here on. We found one triply revised patch, occurring seven days after the release of the second revision.

Notification time in days	Initial revision (20 data points)	Subsequent revision (3 data points)
Maximum	500	62
Minimum	1	1
Average	64.2	22.7
Median	17.5	5
Std deviation	117.0	34.1

Table 4: Analysis of revision data.

Figure 5 presents a histogram over the 20 revised patches of the time from the initial patch release to the time of the announcement of a problem with the patch, while Figure 8 examines the first 30-day period in detail. Figure 6 presents the same data set as a probability at a given time since initial patch release that a patch will be found to be bad, i.e., an empirical plot of $P_{fail}(t)$ from equation 7.

Figure 7 plots the days to resolve an accumulated number of security issues, while Figure 9 examines the first 30-day period more closely. These plots are subtly different from the previous data sets in two ways:

- **Time to resolution:** In the previous graphs, we counted time from when the security patch was announced to the time the patch was announced to be defective. Here, we are measuring to the time the defective patch is resolved. Of the 20 revised patches, 16 provided a revised patch concomitant with the announcement of the patch problem, two had a one-day delay to the release of a revised patch, one had a 97 day delay to the release of a revised patch, and one defective patch was never resolved.
- **No patch ever released:** Of the 136 CVE entries that we surveyed, 24 never had any patch associated with them, and so for these plots, will never be resolved.

Ideally, we would like to be able to overlay Figure 6 with a similar probability plot for “probability of getting hacked at time t past initial disclosure,” or $P_{breach}(t)$. Unfortunately, it is problematic to extract such a probability from Browne, et al.’s data [4] because the numerator (attack incidents) is missing many data points (people who did not bother to report an incident to CERT), and the denominator is huge (the set of all vulnerable nodes on the Internet).

From Honeynet [14] one may extract a $p_{breach}(t)$. Honeynet sought to investigate attacker behavior by placing “honeypot” (deliberately vulnerable) systems on the Internet, and observing the subsequent results. In particular, Honeynet noted that the lifespan of an

older, unpatched Red Hat Linux system containing months-old known vulnerabilities could be as short as a small number of days, as attacker vulnerability scanning tools quickly located and exploited the vulnerable machine. However we note that this probability may not correlate to the system administrator’s site. Site specific factors – site popularity, attention to security updates, vulnerability, etc. – affect the local $p_{breach}(t)$, and as such it must be measured locally.

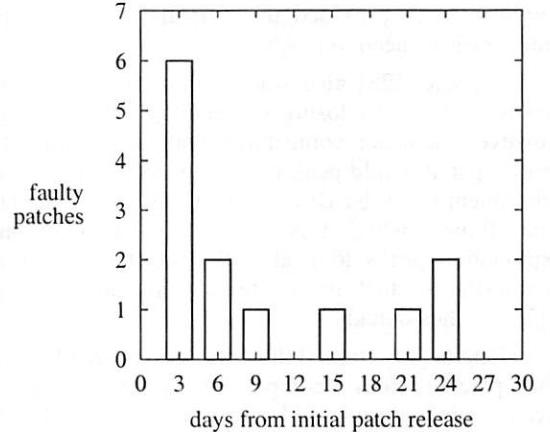


Figure 8: A close-up histogram of the first class interval in Figure 5. It shows the number of faulty initial patch notifications occurring within 30 days of initial patch release.

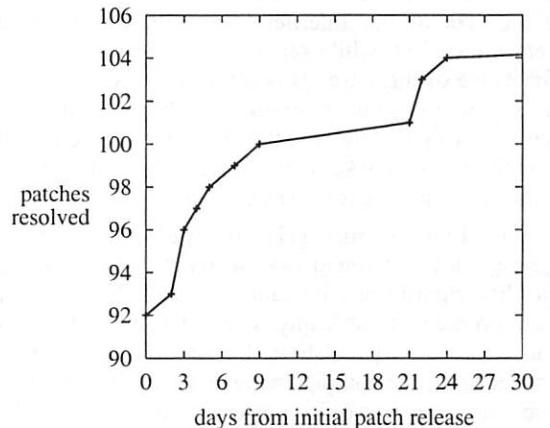


Figure 9: A close-up cumulative graph of the first 30 days in Figure 7. It shows the issue resolution time for those occurring within 30 days of initial patch release.

After determining local probability of a breach (i.e., $p_{breach}(t)$), the administrator should apply Figure 6 to equation 7 to determine the first time t where equation 7 is satisfied. However, since $p_{breach}(t)$ is difficult to compute, the pragmatist may want to observe the knees in the curve depicted in Figures 7 and 9 and apply patches at either ten or thirty days.

Related Work

This paper was inspired by the “Code Red” and “Nimda” worms, which were so virulent that some

analysts conjectured that the security administrators of the Internet could not patch systems fast enough to stop them [20]. Even more virulent worm systems have been devised [28, 26] so the problems of “when to patch?” and “can we patch fast enough?” are very real.

The recent survey of rates of exploitation [4] was critical to our work. In seeking to optimize the trade-off between urgent and cautious patching, it is important to understand both forms of pressure, and Browne, et al. provided the critical baseline of the time-sensitive need to patch.

Schneier [23] also studied rates of exploitation versus time of disclosure of security vulnerabilities. However, Schneier conjectured that the release of a vendor patch would peak the rate of exploitation. The subsequent study by Browne, et al. of CERT incident data above belied this conjecture, showing that exploitation peaks long after the update is released, demonstrating that most site administrators do not apply patches quickly.

Reavis [21] studied the timeliness of vendor-supplied patches. Reavis computed the average “days of recess” (days when a vulnerability is known, but no patch is available) for each of Microsoft, Red Hat Linux, and Solaris. Our clock of “when to patch?” starts when Reavis’ clock of “patch available” stops.

Howard [15] studied Internet security incident rates from 1989 to 1995. He found that, with respect to the size of the Internet, denial-of-service attacks were increasing, while other attacks were decreasing. The cause of these trends is difficult to establish without speculation, but it seems plausible that the exponential growth rate of the Internet exceeded the growth rate of attackers knowledgeable enough to perpetrate all but the easiest (DoS) attacks.

In 1996, Farmer [11] surveyed prominent web hosting sites and found that nearly two-thirds of such sites had significant vulnerabilities, well above the one-third average of randomly selected sites. Again, root causes involve speculation, but it is likely that this resulted from the complex active content that prominent web sites employ versus randomly selected sites. It is also likely that this trend has changed, as e-commerce sites experienced the pressures of security attacks.

In recent work, Anderson [2] presents the viewpoint that many security problems become simpler when viewed through an economic lens. In this paper, we suggest that the system administrator’s failure to patch promptly is actually not a failure, but a rational choice. By analyzing that choice, we are able to suggest a modification to that behavior which addresses the concerns of the party, rather than simply exhorting administrators to patch.

Also worth mentioning is the ongoing study of perception of risk. In McNeil, et al. [17], the authors point out that people told that a medical treatment has a 10% risk of death react quite differently than people

told that 90% of patients survive. It is possible that similar framing issues may influence administrators behavior with respect to security patches.

Discussion

As we performed this study, we encountered several practical issues. Some were practical impediments to the execution of the study, while others were of larger concern to the community of vendors and users. Addressing these issues will both make future research in this area more consistent and valid, and also may improve the situation of the security practitioner.

The first issue is that of setting the values for the constants in our equations, e.g., the cost of breach recovery versus the cost of bad patch recovery, and the probability of a breach for a given site. These values are site-specific, so we cannot ascertain them with any validity:

- A web server that is just a juke box farm of CD-ROMs is not as susceptible to data corruption as an on-line gambling house or a credit bureau, affecting the relative costs of recovery.
- A private corporate server behind a firewall is less likely to be attacked than a public web server hosting a controversial political advocacy page.

We wish to comment that the administrator’s quandary is made worse by vendors who do a poor job of quality assurance on their patches, validating the systems administrator’s decision to not patch. Our ideas can be easily taken by a vendor as advice as to how to improve their patch production process and improve their customer’s security. If the standard deviation of patch failure times is high, then administrators will rationally wait to patch, leaving themselves insecure. Extra work in assurance may pay great dividends. In future work, it would be interesting to examine vendor advance notice (where vendors are notified of security issues ahead of the public) and observe whether the reliability of subsequent patches are more reliable, i.e., do vendors make good use of the additional time.

In collecting data, we noticed but have not yet analyzed a number of trends: Cisco patches failed rarely, while other vendors often cryptically updated their advisories months after issue. The quality of advisories varies widely. We feel it is worth giving kudos to Caldera for the ease with which one can determine that they have issued a new patch [5]. However, they could learn a great deal from some Cisco advisories [8] in keeping detailed advisory revision histories. Red Hat’s advisories included an “issue date,” which we later discovered is actually the first date that they were notified of the issue, not the date they issued the advisory. There has not, to our knowledge, been a paper on “how to write an advisory,” or on the various ways advisories are used.

If one assumes that all problems addressed in this paper relating to buggy patches have been solved,

the administrator must still reliably ascertain the validity of an *alleged* patch. Various forms of cryptographic authentication, such as PGP signatures on Linux RPM packages [3] and digital signatures directly on binary executables [27] can be used. Such methods become essential if one employs automatic patching mechanisms, as proposed by Browne, et al. [4] and provided by services such as Debian apt-get [25], Ximian Red Carpet [29], the Red Hat Network [22], and the automatic update feature in Microsoft Windows XP [18].²

Conclusions

“Never do today what you can put off till tomorrow if tomorrow might improve the odds.”
– Robert Heinlein

The diligent systems administrator faces a quandary: to rush to apply patches of unknown quality to critical systems, and risk resulting failure due to defects in the patch? Or to delay applying the patch, and risk compromise due to attack of a now well-known vulnerability? We have presented models for the pressures to patch early and to patch later, formally modeled these pressures mathematically, and populated the model with empirical data of failures in security patches and rates of exploitation of known flaws. Using these models and data, we have presented a notion of an optimal time to apply security updates. We observe that the risk of patches being defective with respect to time has two knees in the curve at 10 days and 30 days after the patch’s release, making 10 days and 30 days ideal times to apply patches. It is our hope that this model and data will both help to inspire follow-on work and to form a best-practice for diligent administrators to follow.

Author Information

Seth Arnold graduated from Willamette University in 2001 with a B.Sc. in Computer Science, Mathematics, and Religious Studies. He has been a system administrator, has played cryptographer, and is currently employed at WireX Communications in the research group. He can be reached via email at sarnold@wirex.com.

Steve Beattie is employed in the Research Group at WireX Communications and was involved in the development of the StackGuard, SubDomain, RaceGuard and FormatGuard security tools. He received a Masters Degree in Computer Science from the Oregon Graduate Institute, and was previously employed as a Systems Administrator for a Knight-Ridder newspaper. He can be reached via email at steve@wirex.com.

Dr. Crispin Cowan is co-founder and Chief Scientist of WireX, and previously was a Research Assistant Professor at the Oregon Graduate Institute. His research focuses on making existing systems more secure

²We have not verified the cryptographic integrity of any of these automatic update mechanisms.

without breaking compatibility or compromising performance. Dr. Cowan has co-authored 34 refereed publications, including those describing the StackGuard compiler for defending against buffer overflow attacks. He can be reached via email at crispin@wirex.com.

Adam Shostack is currently on sabbatical from his role as Most Evil Genius for Zero-Knowledge systems. Prior to that, he was director of technology for Netect, Inc, where he built vulnerability scanners. He has published on topics including cryptography, privacy, and the economics of security and privacy. He can be reached via email at adam@homeport.org.

Perry Wagle received his M.Sc. in Computer Science at Indiana University in 1995, then dabbled in evolutionary biology until 1997, when he headed to the Oregon Graduate Institute to join the Immunix project’s survivability research. For Immunix, he was, among a number of things, the primary programmer of the first released version of the StackGuard enhancement to GCC. When Immunix spun off into the WireX startup in 1999, he generated a second version of StackGuard, but stayed at OGI to work on the Infosphere project and is still participating in the Timber project there. He recently joined WireX to research various compiler enhancements to building secure Linux distributions, including a third and never-again version of StackGuard. He can be reached via email at wagle@wirex.com.

Chris Wright is one of the maintainers of the Linux Security Module project. He has been with the project since its inception and is helping guide LSM into the mainstream Linux kernel. He is employed by WireX where he gets to do security research and Linux kernel hacking. He can be reached via email at chris@wirex.com.

References

- [1] Anderson, Ross, *Security Engineering: A Guide to Building Dependable Distributed Systems*, John Wiley & Sons, Inc., p. 372, New York, 2001.
- [2] Anderson, Ross, Why Information Security is Hard – An Economic Perspective, *17th Annual Computer Security Applications Conference (ACSAC)*, New Orleans, LA, December 2001.
- [3] Bailey, Ed, *Maximum RPM*, Red Hat Press, 1997.
- [4] Browne, Hilary K., William A. Arbaugh, John McHugh, and William L. Fithen, “A Trend Analysis of Exploitations,” In *Proceedings of the 2001 IEEE Security and Privacy Conference*, pages 214-229, Oakland, CA, <http://www.cs.umd.edu/~waa/pubs/CS-TR-4200.pdf>, May 2001.
- [5] Caldera, Inc., Caldera Security Advisories, <http://www.caldera.com/support/security/>, 2001.
- [6] CERT Coordination Center, CERT Advisory CA-2001-19 “Code Red” Worm Exploiting

- Buffer Overflow In IIS Indexing Service DLL, <http://www.cert.org/advisories/CA-2001-19.html>, August 2001.
- [7] Christey, Steven M., An Informal Analysis of Vendor Acknowledgement of Vulnerabilities, Bugtraq Mailing List, <http://www.securityfocus.com/cgi-bin/archive.pl?id=1&mid=168287>, March 2001.
- [8] Cisco Systems, Inc., Security Advisory: Cisco Content Services Switch Vulnerability, <http://www.cisco.com/warp/public/707/arrowpoint-clifilesystem-pub.shtml>, April 2001.
- [9] Cowan, Crispin, Steve Beattie, Calton Pu, Perry Wagle, and Virgil Gligor, SubDomain: Parsimonious Server Security, *USENIX 14th Systems Administration Conference (LISA)*, New Orleans, LA, December 2000.
- [10] Dittrich, Dave, *The Forensic Challenge*, <http://project.honeynet.org/challenge/>, January 2001.
- [11] Farmer, Dan, *Shall We Dust Moscow? Security Survey of Key Internet Hosts & Various Semi-Relevant Reflections*, <http://www.fish.com/survey/>, December 1996.
- [12] Goldberg, Ian, David Wagner, Randi Thomas, and Eric Brewer, "A Secure Environment for Untrusted Helper Applications," *6th USENIX Security Conference*, San Jose, CA, July 1996.
- [13] Hinton, Heather M., Crispin Cowan, Lois Delcambre, and Shawn Bowers, "SAM: Security Adaptation Manager," *Annual Computer Security Applications Conference (ACSAC)*, Phoenix, AZ, December 1999.
- [14] The Honeynet Project, *Know Your Enemy: Revealing the Security Tools, Tactics and Motives of the BlackHat Community*, Addison Wesley, Boston, 2002.
- [15] Howard, John D., *An Analysis of Security Incidents on the Internet 1989 – 1995*, Ph.D. thesis, Carnegie Mellon University, <http://www.cert.org/research/JHThesis/Start.html>, October 1997.
- [16] Martin, Robert A., "Managing Vulnerabilities in Networked Systems," *IEEE Computer Society COMPUTER Magazine*, pp. 32-38, <http://cve.mitre.org/>, November 2001.
- [17] McNeil, B. J., S. G. Pauker, H. C. Sox, and A. Tversky, "On the Elicitation of Preferences for Alternative Therapies," *New England Journal of Medicine*, No. 306, pp. 1259-1262, 1982.
- [18] Microsoft, *Automatic Update Feature in Windows XP*, <http://support.microsoft.com/directory/article.asp?ID=KB;EN-US;Q294871>, October 2001.
- [19] Microsoft Corporation, *Microsoft Security Bulletin MS01-031 Predictable Name Pipes Could Enable Privilege Elevation via Telnet*, <http://www.microsoft.com/technet/security/bulletin/MS01-031.asp?frame=true>, June 2001.
- [20] Pescatore, John, *Nimda Worm Shows You Can't Always Patch Fast Enough*, <http://www.gartner.com/DisplayDocument?id=340962>, September 2001.
- [21] Reavis, Jim, *Linux vs. Microsoft: Who Solves Security Problems Faster?*, [no longer available on the web], January 2000.
- [22] Red Hat, Inc., *Red Hat Update Agent*, <https://rhn.redhat.com/help/sm/up2date.html>, 2001.
- [23] Schneier, Bruce, *Full Disclosure and the Window of Exposure*, <http://www.counterpane.com/cryptogram-0009.html#1>, September 2000.
- [24] Shakespeare, William, *Hamlet*, Act I, Scenes 3 and 4, F. S. Crofts & Co., New York, 1946.
- [25] Silva, Gustavo Noronha, *Debian APT Howto*, <http://www.debian.org/doc/manuals/apt-howto/index.en.html>, September 2001.
- [26] Staniford, Stuart, Gary Grim, and Roelof Jonkman, *Flash Worms: Thirty Seconds to Infect the Internet*, <http://www.silicondefense.com/flash/>, August 2001.
- [27] van Doorn, Leendert, Gerco Ballintijn, and William A. Arbaugh, *Signed Executables for Linux*, Technical Report UMD CS-TR-4259, University of Maryland, 2001.
- [28] Weaver, Nicholas C., *Warhol Worms: The Potential for Very Fast Internet Plagues*, <http://www.cs.berkeley.edu/~nweaver/warhol.html>, August 2001.
- [29] Ximian Inc., *Ximian Red Carpet Automated Software Maintenance and Version Management*, 2001.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

Member Benefits

- Free subscription to *;login:*, the Association's magazine, published eight times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and Open Source, book and software reviews, summaries of sessions at USENIX conferences, and Standards Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to *;login:* on the USENIX Web site.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, on the USENIX Web site.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as security, Linux, Internet technologies and systems, operating systems, and Windows—as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Savings on a variety of products, books, software, and periodicals: see <http://www.usenix.org/membership/specialdisc.html> for details.

Supporting Members of the USENIX Association

Atos Origin B.V.	Sendmail, Inc.
Freshwater Software	Sun Microsystems, Inc.
Interhack Corporation	Sybase, Inc.
Microsoft Research	Taos: The Sys Admin Company
Motorola Australia Software Centre	UUNET Technologies, Inc.
OSDN	Ximian, Inc.

Supporting Members of SAGE

Certainty Solutions	Microsoft Research
Collective Technologies	OSDN
ESM Services, Inc.	Ripe NCC
Freshwater Software	

For more information about membership, conferences, or publications,

see <http://www.usenix.org/>

or contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA
Phone: 510-528-8649 Fax: 510-548-5738 Email: office@usenix.org

ISBN 1-931971-03-X